ARCoSS

LNCS 6950

David Peleg (Ed.)

Distributed Computing

25th International Symposium, DISC 2011
Rome, Italy, September 2011
Proceedings

Springer

David Peleg (Ed.)

Distributed Computing

25th International Symposium, DISC 2011
Rome, Italy, September 20-22, 2011
Proceedings

Springer

Volume Editor

David Peleg
Weizmann Institute of Science
Department of Computer Science
Rehovot, Israel
E-mail: david.peleg@weizmann.ac.il

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24099-7 e-ISBN 978-3-642-24100-0
DOI 10.1007/978-3-642-24100-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011936639

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

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

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 2011, the 25th International Symposium on Distributed Computing, held during September 20–22, 2011 in Rome, Italy.

There were 136 regular papers submitted to the symposium (in addition to a large number of abstract-only submissions). The Program Committee selected 31 contributions out of the 136 full paper submissions for regular presentations at the symposium. Each presentation was accompanied by a paper of up to 15 pages in this volume. Every submitted paper was read and evaluated by at least three members of the Program Committee. The committee was assisted by about 190 external reviewers. The Program Committee made its final decisions during an electronic meeting held on June 1–22, 2011. Revised and expanded versions of several selected papers will be considered for publication in a special issue of the journal *Distributed Computing*.

The Best Paper Award of DISC 2011 was presented to Pierre Fraigniaud, Sergio Rajsbaum and Corentin Travers for the paper "Locality and Checkability in Wait-Free Computing."

The Best Student Paper Award of DISC 2011 was presented to Michael Hakimi and Adam Morrison for the paper "Fast and Scalable Rendezvousing," co-authored with Yehuda Afek.

The Program Committee also considered about 30 papers for brief announcements, among the papers that were submitted as brief announcements, as well as the regular submissions that generated substantial interest from the members of the committee, but that could not be accepted for regular presentations. This volume contains 11 brief announcements. Each two-page announcement presents ongoing work or recent results, and it is expected that these results will appear as full papers in other conferences or journals.

The program also featured three invited lectures, presented by Dahlia Malkhi (Microsoft Research), Andrzej Pelc (Université du Québec en Outaouais), and Peter Widmayer (ETH - Swiss Federal Institute of Technology Zurich). Papers summarizing the contents of these invited lectures are included in these proceedings.

In addition, there were two tutorials offered in the program. The first, presented by Yoram Moses, was titled "Knowledge Strikes Again," and dealt with using knowledge for reasoning about distributed computation. The second tutorial, presented by Christian Cachin, was titled "From Reliable to Secure

Distributed Programming" and concerned making distributed programs Byzantine fault-tolerant.

Last, but not least, the symposium program also featured a celebration in honor of Nicola Santoro's 60th birthday.

Five workshops were co-located with the DISC symposium this year: the Third Workshop on Theoretical Aspects of Dynamic Distributed Systems (TADDS), organized by Alexander Shvartsman and Roberto Baldoni, on September 19; the workshop Toward Evolutive Routing Algorithms for Scale-Free/Internet-Like Networks (TERANET), organized by David Ilcinkas and Dimitri Papadimitriou, on September 19; the First International Workshop on Algorithms and Models for Distributed Event Processing (AlMoDEP), organized by Leonardo Querzoni and Luigi Laura, on September 19; the TransForm Workshop on the Theory of Transactional Memory (TransForm WTTM 2011/Euro-TM Workshop), organized by Petr Kuznetsov and Srivatsan Ravi, on September 22 and 23; and DISC's Social Network Workshop (DISC's SON), organized by Alessandro Panconesi, on September 23.

DISC 2011 acknowledges the use of the EasyChair system for handling submissions, managing the review process, and compiling these proceedings.

September 2011 David Peleg

Symposium Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for 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 biannual 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

David Peleg	Weizmann Institute of Science, Israel

Program Committee

Marcos K. Aguilera	Microsoft Research, USA
Roberto Baldoni	Università di Roma La Sapienza, Italy
Konstantin Busch	Louisiana State University, USA
Keren Censor-Hillel	MIT, USA
Ioannis Chatzigiannakis	Computer Technology Institute, Greece
Danny Dolev	Hebrew University, Israel
Faith Ellen	University of Toronto, Canada
Yuval Emek	ETH Zurich, Switzerland
Sándor Fekete	TU Braunschweig, Germany
Luisa Gargano	University of Salerno, Italy
Maurice Herlihy	Brown University, USA
David Ilcinkas	CNRS and University of Bordeaux, France
Anne-Marie Kermarrec	INRIA, Rennes, France
Adrian Kosowski	INRIA, University of Bordeaux, France
Toshimitsu Masuzawa	Osaka University, Japan
Gopal Pandurangan	NTU Singapore and Brown University, USA
Maria Potop-Butucaru	University of Paris 6, France
Michael Spear	Lehigh University, USA
Philipp Woelfel	University of Calgary, Canada

Steering Committee

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

Local Organization

Carola Aiello	Università di Roma La Sapienza
Roberto Baldoni (Chair)	Università di Roma La Sapienza
Silvia Bonomi	Università di Roma La Sapienza
Gabriella Caramagno	Università di Roma La Sapienza
Adriano Cerocchi	Università di Roma La Sapienza
Leonardo Querzoni	Università di Roma La Sapienza

External Reviewers

Ittai Abraham
H.B. Acharya
Yehuda Afek
Dan Alistarh
Dimitris Amaxilatis
Aris Anagnostopoulos
Emmanuelle Anceaume
Athanasios Antoniou
Anish Arora
James Aspnes
Hagit Attiya
John Augustine
Vincenzo Auletta
Chen Avin
Athanasios Bamis
Surender Baswana
Vibhor Bhatt
Carlo Blundo
Nicolas Bonichon
Francois Bonnet
Silvia Bonomi
Trevor Brown
Niv Buchbinder
Armando Castaneda
Boaz Catane
Jérémie Chalopin
Shiri Chechik
Jen-Yeu Chen
Bogdan Chlebus
Ferdinando Cicalese
Allen Clement
Colin Cooper
Gennaro Cordasco
David Coudert
Paolo D'Arco
Mike Dahlin
Atish Das Sarma
Shantanu Das
Gianluca De Marco
Roberto De Prisco
Sylvie Delaët
Carole Delporte-Gallet
Bilel Derbel
Stéphane Devismes
Josep Diaz
David Dice
Yann Disser
Stefan Dulman

Chinmoy Dutta
Michael Elkin
Lionel Eyraud-Dubois
Panagiota Fatourou
Paola Flocchini
Davide Frey
Tobias Friedrich
Satoshi Fujita
Emanuele Guido Fusco
Jie Gao
Vijay Garg
Leszek Gasieniec
Mohsen Ghaffari
George Giakkoupis
Yiannis Giannakopoulos
Seth Gilbert
Wojciech Golab
Vincent Gramoli
Rachid Guerraoui
Maxim Gurevich
Joseph Halpern
Nicolas Hanusse
Henning Hasemann
Maryam Helmi
Danny Hendler
Ted Herman
Amir Herzberg
Lisa Higham
Martin Hirt
Ezra Hoch
Stephan Holzer
Mohammad Reza Hoseiny Farahabady
Michiko Inoue
Taisuke Izumi
Navendu Jain
Prasad Jayanti
Colette Johnen
Hirotsugu Kakugawa
Erez Kantor
Yoshiaki Katayama
Barbara Keller
Barbara Kempkes
Ralf Klasing
Panagiotis Kokkinos
Spyros Kontogiannis
Guy Korland
Amos Korman
Miroslaw Korzeniowski
Eleftherios Kosmas
Dariusz Kowalski
Fabian Kuhn
Ranjit Kumaresan
Lukasz Kuszner
Tobias Langner
Mikel Larrea
Nicolas Le Scouarnec
Christoph Lenzen
Giorgia Lodi
Zvi Lotker
Victor Luchangco
Nancy Lynch
Frédéric Majorczyk
Dahlia Malkhi
Yishay Mansour
Alex Matveev
Petar Maymounkov
James McLurkin
George B. Mertzios
Othon Michail
Alessia Milani
Afshin Moin
Pat Morin
Luca Moscardelli
Georgios Mylonas
Danupon Nanongkai
Alfredo Navarra
Stavros Nikolaou
Fukuhito Ooshita
Rotem Oshman
Saurav Pandit
Behrooz Parhami
Gahyun Park
Merav Parter
Andreas Pavlogiannis
Ami Paz
Andrzej Pelc
Alberto Pettarin
Laurence Pilard
Andreas Prell
Apostolos Pyrgelis

Vivien Quema
Leonardo Querzoni
Tomasz Radzik
Anisur Rahaman
Sergio Rajsbaum
Ingy Ramzy
Tzachy Reinman
Adele Rescigno
Peter Robinson
Liam Roditty
Matthieu Roy
Eric Ruppert
Srikanth Sastry
Ignasi Sau
Christian Scheideler
Bill Scherer
Elad Michael Schiller
Christian Schindelhauer
Stefan Schmid
Christiane Schmidt
Marco Serafini
Gokarna Sharma
Nir Shavit
Alexander Shraer
Jasmin Smula
Shay Solomon
Philipp Sommer
Paul Spirakis
Srivathsan Srinivasagopalan
Zhifeng Sun
Jukka Suomela
Francois Taiani
Srikanta Tirthapura
Sébastien Tixeuil
Corentin Travers
Amitabh Trehan
Vasileios Trigonakis
Sara Tucci-Piergiovanni
Shimrit Tzur
Ugo Vaccaro
Yevgeniy Vahlis
Mathieu Valero
Robbert Van Renesse
Fabio Vandin
Laurent Viennot
Anil Vullikanti
Koichi Wada
Jennifer Welch
Peter Widmayer
Avani Wildani
Juerg Wullschleger
Akka Zemmari
Oles Zhulyn

Sponsoring Organizations

European Association for Theoretical Computer Science

Università di Roma 'La Sapienza'

CINI: Consorzio Interuniversitario Nazionale per l'Informatica

Microsoft Research

Table of Contents

Invited Lecture & Best Student Paper: Rendezvous (Session 1a)

DISC 2011 Invited Lecture: Deterministic Rendezvous in Networks: Survey of Models and Results 1
Andrzej Pelc

Fast and Scalable Rendezvousing 16
Yehuda Afek, Michael Hakimi, and Adam Morrison

Distributed Graph Algorithms (Session 1b)

Beeping a Maximal Independent Set 32
Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn

Trading Bit, Message, and Time Complexity of Distributed Algorithms 51
Johannes Schneider and Roger Wattenhofer

Combinatorial Algorithms for Distributed Graph Coloring 66
Leonid Barenboim and Michael Elkin

Physical Expander in Virtual Tree Overlay 82
Taisuke Izumi, Maria Gradinariu Potop-Butucaru, and Mathieu Valero

Shared Memory (Session 1c)

Sub-logarithmic Test-and-Set against a Weak Adversary 97
Dan Alistarh and James Aspnes

Tight Space Bounds for ℓ-Exclusion 110
Gadi Taubenfeld

SMV: Selective Multi-Versioning STM 125
Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar

Brief Announcements I (Session 1d)

Brief Announcement: Leaderless Byzantine Paxos 141
Leslie Lamport

Brief Announcement: When You Don't Trust Clients: Byzantine Proposer Fast Paxos 143
Keith Marzullo, Hein Meling, and Alessandro Mei

Brief Announcement: On the Meaning of Solving a Task with a Failure Detector 145
Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov

Brief Announcement: Algorithmic Mechanisms for Internet-Based Computing under Unreliable Communication 147
Evgenia Christoforou, Antonio Fernández Anta, Chryssis Georgiou, and Miguel A. Mosteiro

Fault-Tolerance and Security (Session 1e)

Maximum Metric Spanning Tree Made Byzantine Tolerant 150
Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil

Performing Dynamically Injected Tasks on Processes Prone to Crashes and Restarts 165
Chryssis Georgiou and Dariusz R. Kowalski

Leakage-Resilient Coin Tossing 181
Elette Boyle, Shafi Goldwasser, and Yael Tauman Kalai

Brief Announcements II (Session 1f)

Brief Announcement: Composition Games for Distributed Systems: The EU Grants games 197
Shay Kutten, Ron Lavi, and Amitabh Trehan

Brief Announcement: Distributed Approximations for the Semi-matching Problem 200
Andrzej Czygrinow, Michal Hanćkowiak, Krzysztof Krzywdziński, Edyta Szymańska, and Wojciech Wawrzyniak

Brief Announcement: Opportunistic Information Dissemination in Mobile Ad-Hoc Networks: Adaptiveness vs. Obliviousness and Randomization vs. Determinism 202
Martín Farach-Colton, Antonio Fernández Anta, Alessia Milani, Miguel A. Mosteiro, and Shmuel Zaks

Brief Announcement: Bridging the Theory-Practice Gap in Multi-Commodity Flow Routing 205
Siddhartha Sen, Sunghwan Ihm, Kay Ousterhout, and Michael J. Freedman

Invited Lecture: Paxos Plus (Session 2a)

DISC 2011 Invited Lecture by Dahlia Malkhi: Going beyond Paxos 208
Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, and Ted Wobber

Byzantizing Paxos by Refinement 211
Leslie Lamport

Wireless (Session 2b)

Unbounded Contention Resolution in Multiple-Access Channels 225
Antonio Fernández Anta, Miguel A. Mosteiro, and Jorge Ramón Muñoz

Deterministic and Energy-Optimal Wireless Synchronization 237
Leonid Barenboim, Shlomi Dolev, and Rafail Ostrovsky

Leveraging Channel Diversity to Gain Efficiency and Robustness for Wireless Broadcast 252
Shlomi Dolev, Seth Gilbert, Majid Khabbazian, and Calvin Newport

Leader Election Using Loneliness Detection 268
Mohsen Ghaffari, Nancy Lynch, and Srikanth Sastry

Network algorithms I (Session 2c)

Optimal Random Sampling from Distributed Streams Revisited 283
Srikanta Tirthapura and David P. Woodruff

Parsimonious Flooding in Geometric Random-Walks (Extended Abstract) 298
Andrea E.F. Clementi and Riccardo Silvestri

Misleading Stars: What Cannot Be Measured in the Internet? 311
Yvonne Anne Pignolet, Stefan Schmid, and Gilles Tredan

Brief Announcements III (Session 2d)

Brief Announcement: A Randomized Algorithm for Label Assignment in Dynamic Networks 326
Meg Walraed-Sullivan, Radhika Niranjan Mysore, Keith Marzullo, and Amin Vahdat

Brief Announcement: $\Delta\Omega$: Specifying an Eventual Leader Service for Dynamic Systems 328
Mikel Larrea and Michel Raynal

Brief Announcement: The BG-Simulation for Byzantine Mobile Robots 330
Taisuke Izumi, Zohir Bouzid, Sébastien Tixeuil, and Koichi Wada

Invited Lecture & Best Paper: Aspects of Locality (Session 3a)

DISC 2011 Invited Lecture: Polygon Reconstruction with Little Information: An Example for the Power of Simple Micro-robots 332
Peter Widmayer

Locality and Checkability in Wait-Free Computing 333
Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers

Consensus (Session 3b)

The Contest between Simplicity and Efficiency in Asynchronous Byzantine Agreement 348
Allison Lewko

Randomized Consensus in Expected $O(n^2)$ Total Work Using Single-Writer Registers 363
James Aspnes

Structured Derivation of Semi-Synchronous Algorithms 374
Hagit Attiya, Fatemeh Borran, Martin Hutle, Zarko Milosevic, and André Schiper

Byzantine Agreement Using Partial Authentication 389
Piyush Bansal, Prasant Gopal, Anuj Gupta, Kannan Srinathan, and Pranav Kumar Vasishta

Network algorithms II (Session 3c)

On Approximate Distance Labels and Routing Schemes with Affine Stretch 404
Ittai Abraham and Cyril Gavoille

The Complexity of Data Aggregation in Directed Networks 416
Fabian Kuhn and Rotem Oshman

Black Hole Search with Finite Automata Scattered in a Synchronous Torus 432
Jérémie Chalopin, Shantanu Das, Arnaud Labourel, and Euripides Markou

Synchronous Rendezvous for Location-Aware Agents 447
Andrew Collins, Jurek Czyzowicz, Leszek Gąsieniec, Adrian Kosowski, and Russell Martin

Concurrency (Session 3d)

Toward a Formal Semantic Framework for Deterministic Parallel Programming 460
Li Lu and Michael L. Scott

CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention ... 475
Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman

Oblivious Collaboration 489
Yehuda Afek, Yakov Babichenko, Uriel Feige, Eli Gafni, Nati Linial, and Benny Sudakov

Author Index 505

DISC 2011 Invited Lecture: Deterministic Rendezvous in Networks: Survey of Models and Results

Andrzej Pelc*

Département d'informatique, Université du Québec en Outaouais,
Gatineau, Québec J8X 3X7, Canada
pelc@uqo.ca

Abstract. Two or more mobile entities, called *agents* or *robots*, starting at distinct initial positions in some environment, have to meet. This task is known in the literature as *rendezvous*. Among many alternative assumptions that have been used to study the rendezvous problem, two most significantly influence the methodology appropriate for its solution. The first of these assumptions concerns the environment in which the mobile entities navigate: it can be either a terrain in the plane, or a network modeled as an undirected graph. In the case of networks, methods and results further depend on whether the agents have the ability to mark nodes in some way. The second assumption concerns the way in which the entities move: it can be either deterministic or randomized. In this paper we survey models and results concerning deterministic rendezvous in networks, where agents cannot mark nodes.

Keywords: mobile agent, rendezvous, deterministic, network, graph.

1 Introduction

Two or more mobile entities, starting at distinct initial positions, have to meet. This task, called *rendezvous*, has numerous applications in domains ranging from human interaction and animal behavior to programming of autonomous mobile robots and software agents. Algorithmic problems requiring accomplishing a rendezvous task in a human-made environment come up mainly in two situations. The first of them concerns autonomous mobile robots that start in different locations of a planar terrain or a labyrinth and have to meet, e.g., in order to exchange information obtained while exploring the terrain and coordinate further actions. The second situation concerns software agents, i.e., mobile pieces of software that travel in a communication network in order to perform maintenance of its components or to collect data distributed in nodes of the network. Such software agents also need to meet periodically, in order to exchange collected data and plan further moves. Since rendezvous algorithms do not depend on the physical nature of the mobile entities, but only on their characteristics,

* Supported in part by NSERC discovery grant and by the Research Chair in Distributed Computing of the Université du Québec en Outaouais.

such as perceiving capabilities, memory, mobility and on the nature of the environment, throughout this paper mobile entities that have to meet will be called by the generic name of *agents*, regardless of whether, in any particular application, these are people, animals, mobile robots, or software agents. In the case of more than two agents, the rendezvous problem is sometimes called *gathering.* For the sake of uniformity, we will call it rendezvous also in this case.

Since agents that have to meet are not coordinated by any central monitor and have to make moving decisions autonomously, the rendezvous problems belong naturally to the field of distributed computing. Among many alternative assumptions that have been used to study the rendezvous problem, two most significantly influence the methodology appropriate for its solution. The first of these assumptions concerns the environment in which the agents navigate: it can be either a terrain in the plane, or a network modeled as an undirected graph. While rendezvous in the plane calls mostly for geometric considerations, see, e.g., [5], the network scenario involves mainly methods coming from graph theory. The second assumption concerns the way in which the agents move: it can be either deterministic or randomized. More precisely, in any deterministic scenario, the initial positions of the agents are chosen by an adversary which models the worst-case situation, and each move of the agent is determined only by its current history that may include the identity of the agent (if any), and the part of the environment that the agent has seen to date. By contrast, in a randomized scenario, initial positions of the agents are chosen at random and their moves may also involve coin tosses. The cost of rendezvous is also different in both scenarios: while in deterministic rendezvous the concern is with the worst-case cost (usually defined as the time or the length of the agents' trajectories until rendezvous), in the randomized scenario it is the expected value of these quantities. In both cases the problem is often to minimize the worst case (resp. expected) cost. Deterministic rendezvous problems usually require combinatorial methods, while randomized rendezvous often calls for analytic tools.

In this paper we survey results concerning deterministic rendezvous in networks, thus from the outset we restrict attention only to agents navigating in networks and only to agents whose entire behavior is deterministic. Moreover, we concentrate on the scenario where agents cannot mark visited nodes in any way, neither by writing on whiteboards nor by dropping tokens.

This decision has three reasons. The first of them is the necessity of choosing only a part of the vast body of the literature on rendezvous, due to space limitations. The second reason is the wish to choose a body of problems whose solutions are methodologically homogeneous. This is the main rationale behind restricting attention to the network environment. Finally, we wanted to avoid duplication of existing surveys on rendezvous. There are four main such surveys. Chronologically the first of them is [1], almost entirely contained in the second part of the excellent book [2], the latter being by far the most comprehensive compendium to date on rendezvous problems. Both [1] and [2] concern randomized rendezvous, which is the main reason of our restriction to the deterministic case. The third survey is [20]. While its scope is large, the authors concentrate mainly on presenting rendezvous models and compare their underlying assumptions. Finally, the recent book [19] deals mostly with rendezvous problems on the ring, only briefly mentioning other network topologies in this context. Also most of the attention in [19] is devoted to the scenario where visited nodes can be marked by agents, usually using tokens. The aim of the present survey is different. It differs from [1,2] by concentrating on deterministic rather than on randomized settings; it differs from [20] by the level of details in treating the rendezvous problem: besides presenting various models under which rendezvous is studied, we want to report precisely the results obtained under each of them, discussing how varying assumptions influence feasibility and complexity of rendezvous under different scenarios. Finally, the present survey differs from [19] by discussing many different topologies, including arbitrary, even unknown graphs, rather than concentrating on a particular type of networks. Also, unlike [19], we focus on the scenario where marking of nodes is not allowed.

The following assumptions are common to all papers that we will survey. The first is modeling the network as a simple undirected connected graph, whose nodes represent processors, computers or stations of a communication network, or crossings of corridors of a labyrinth, depending on the application, and links represent communication channels in a communication network, or corridors in a labyrinth. Undirectedness of the graph captures the fact that agents may move in both directions along each link, the assumption that the graph is simple (no self-loops or multiple edges) is motivated by most of the realistic examples, and connectivity of the graph is a necessary condition on feasibility of rendezvous when starting from any initial positions.

There are two other assumptions common to all surveyed papers. One is the anonymity of the underlying network: the absence of distinct names of nodes that can be perceived by the navigating agents. There are two reasons for seeking rendezvous algorithms that do not assume knowledge of node identities. The first one is practical: while nodes may indeed have different labels, they may refrain from informing the agents about them, e.g., for security reasons, or limited sensory capabilities of agents may prevent them from perceiving these names. The other reason for assuming anonymity of the network is that if distinct names of nodes can be perceived by the agents, they can follow an algorithm which guides each of them to the node with the smallest label and stop. Thus the rendezvous problem reduces to graph exploration.

The last common assumption concerns port numbers at each node: a node of degree d has ports $0, 1, \dots, d-1$ corresponding to the incident edges. Ports at each node are seen by an agent visiting this node, but there is no coherence assumed between port labelings at different nodes. The reason for assuming the existence of port labelings accessible to agents is the following. If an agent is unable to locally distinguish ports at a node, it may even be 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. Consequently, agents initially located at two nodes of degree at least 3 might never be able to meet. Note that the absence of port numbers need not preclude rendezvous, if agents are allowed to take periodic snapshots of the entire network, as in [16,17].

2 Taxonomy of Rendezvous Problems

The main problem that has to be solved in order to make deterministic rendezvous possible in large classes of networks is breaking symmetry. To see why this is necessary, consider a highly symmetric network, such as an oriented ring or an oriented torus. In the first case we mean a ring in which ports at all nodes are labeled as follows: 0 the clockwise port and 1 the counterclockwise port. Similarly, in an oriented torus ports North, East, South and West at each node are labeled 0,1,2,3, respectively. Consider two identical agents starting at distinct nodes of one of these networks and running the same deterministic algorithm. If agents are unable to mark nodes in any way, it is easy to see that they will never meet. Indeed, at all times they will use the port (at their respective current nodes) having the same label (as their history is the same and the algorithm is deterministic), and hence the distance between them will be always the same. (Notice that the situation would be much different, if randomization were allowed. In this case symmetry can be efficiently broken with high probability using coin tosses to determine the next port to be used.)

In the deterministic scenario there are three ways to break symmetry. The first is by distinguishing the agents: each of them has a label and the labels are different. Each agent knows its label, but we do not need to assume that it knows the label of the other agent. (If it does, then the solution is the well-known algorithm Wait For Mommy: the agent with the smaller label stays idle, while the other one explores the graph in order to find it.) Both agents use the same *parametrized* algorithm with the agent's label as the parameter. To see how this can help, consider two agents that have to meet in an oriented ring of known size n. As mentioned above, if agents are anonymous (and marking nodes is disallowed), rendezvous is impossible. Now assume that agents have distinct labels L_1 and L_2. A simple rendezvous algorithm is:

– Make L tours of the ring, where L is your label, and stop.

Then the agent with larger label will make at least one full tour of the ring while the other one is already inert, thus guaranteeing rendezvous.

The second way of breaking symmetry is marking nodes, either by allowing agents to drop tokens on visited nodes, or by using whiteboards at nodes, on which agents can write and from which they can read information. As mentioned above, this scenario will not be discussed in this survey.

Finally, the third way of breaking symmetry is by exploiting either non-symmetries of the network itself, or the differences of the initial positions of the agents, even in a symmetric network. This method is usable only for some classes of networks, as either the network must have distinguishable nodes that play the role of "focal points" or the initial positions of agents have to be "non-symmetric" (the precise meaning of this condition will be defined later). On the other hand, the method is applicable even when agents are identical and nodes cannot be marked. As a simple example of the application of this method, consider a n-node line with two identical agents. If n is odd, then the line contains a central node that both agents can identify and meet at this node. If n is even, (and even when the port labelings are symmetric with respect to the axis of symmetry of the line) but the initial positions of the agents have different distances from their closest extremity, then the following algorithm works:

– Compute your distance d from the closest extremity of the line, then traverse the line d times and stop.

For the same reasons as before, this algorithm guarantees rendezvous, whenever the initial positions of the agents are not symmetrically situated. On the other hand, if they are symmetric (and port labelings are symmetric as well), then it is easy to see that rendezvous is impossible. A variation of the method of exploiting asymmetries of the initial configuration of agents occurs when there are more than two agents that are able to take periodic snapshots of the network: this scenario will be explained in details in Section 4.

As seen above, a scenario under which the rendezvous problem is studied must specify the way in which symmetry is broken. This is usually connected with the assumptions concerning the capability of agents to perceive and interact with the environment. The weakest assumptions usually made in this respect are that an agent entering a node perceives the degree of the visited node and the port number by which it enters the node (however, the second of these assumptions is sometimes dropped, see, e.g., [22]), and that it can read the port numbers at the visited nodes in order to use the one that the algorithm indicates as the exit port. As mentioned above, in some cases, much more powerful perception capabilities are assumed: it is supposed that an agent may periodically take a snapshot of the network, seeing where other agents are situated.

Another assumption that significantly influences feasibility and efficiency of rendezvous is the amount of memory with which agents are equipped. It is either assumed that this memory is unbounded and agents are modeled as Turing machines, or that the memory is bounded, in which case the model of input/output automata (finite state machines) is usually used.

A different type of assumptions concerns the way in which agents move. Here an important dichotomy is between the synchronous and asynchronous scenarios. In the synchronous scenario agents move from node to node in synchronous rounds and the meeting has to occur at a node. Asynchrony is captured in two different ways. The precise definitions will be given later, but the general idea is the following. In one model, an adversary decides when each agent moves, but the move itself is instantaneous, thus it is possible to require meeting at a node, as previously. In the second model, the agent chooses an edge but the adversary determines the actual walk of the agent on this edge and can, e.g., speed up or slow down the agent. Under this scenario it may be impossible to meet at a node, and thus the requirement is relaxed to that of meeting at a node or inside an edge.

3 Synchronous Rendezvous

In this section we focus on the synchronous setting. Agents move in synchronous steps. In every step, an agent may either remain at the same node or move to an adjacent node. Rendezvous means that all agents are at the same node in the same step. Agents that cross each other when moving along the same edge, do not notice this fact. Two scenarios are considered: *simultaneous startup*, when both agents start executing the algorithm at the same time, and *arbitrary delay*, when starting times are arbitrarily decided by an adversary. In the former case, agents know that starting times are the same, while in the latter case, they are not aware of the difference between starting times, and each of them starts executing the rendezvous algorithm and counting steps since its own startup. The agent that starts earlier and happens to visit the starting node of the later agent *before* the startup of this later agent, is not aware of this fact, i.e, it is assumed that agents are created at their startup time and not waiting at the node before it.

In [10] (whose journal version was published in 2006, but which is based on two earlier conference papers published in 2003 and 2004), rendezvous of two agents is considered and it is indicated that all results can be generalized to an arbitrary number of agents. It is assumed that agents have different labels, which are positive integer numbers, and each agent knows its own label (which is a parameter of the common algorithm that they use), but is unaware of the label of the other agent. In general, agents do not know the topology of the graph in which they have to meet. Hence an agent, currently located at a node, does not know the other endpoints of yet unexplored incident edges. If the agent decides to traverse such a new edge (which means that it chooses a port number not yet taken at this node), the choice of the actual edge belongs to an adversary, in order to capture the worst-case performance. It is assumed that the agents have unlimited memory and the authors aim at optimizing the cost of rendezvous. For a given initial location of agents in a graph, this cost is defined as the worst-case number of steps since the startup of the later agent until rendezvous is achieved, where the worst case is taken over all adversary decisions, whenever an agent decides to explore a new edge adjacent to a currently visited node, and over all possible startup times (decided by an adversary), in the case of the arbitrary delay scenario.

The following notation is used. The labels of the agents are L_1 and L_2. The smaller of the two labels is denoted by l. The delay (the difference between startup times of the agents) is denoted by τ, n denotes the number of nodes in the graph, and D – the distance between initial positions of agents.

The authors start by introducing the problem in the relatively simple case of rendezvous in trees. They show that rendezvous can be completed at cost $O(n + \log l)$ on any n-node tree, even with arbitrary delay. It is also shown that for some trees this complexity cannot be improved, even with simultaneous startup. The class of trees is relatively easy from the point of view of rendezvous. Indeed, any tree has either a central node or a central edge. In the first case this node plays the role of the “focal point” and rendezvous can be accomplished at linear cost. In the second case, rendezvous is slightly more complicated and, after identifying the central edge, it reduces to rendezvous in the two-node graph. It is this case that is responsible for the $O(\log l)$ term in the cost complexity.

As soon as the graph contains cycles, another technique has to be applied. The authors continue the study by concentrating on the simplest class of such graphs, i.e., rings. They prove that, with simultaneous startup, the optimal cost of rendezvous on any ring is $\Theta(D \log l)$. They construct an algorithm achieving rendezvous with this complexity and show that, for any distance D, it cannot be improved.

With an arbitrary delay, $\Omega(n + D \log l)$ is a lower bound on the cost required for rendezvous in a n-node ring. Under this scenario, two rendezvous algorithms for the ring are presented in [10]: an algorithm with cost $O(n \log l)$, for known n, and an algorithm with cost $O(l\tau + ln^2)$, if n is unknown.

For arbitrary connected graphs, the main contribution of [10] is a deterministic rendezvous algorithm with cost polynomial in n, τ and $\log l$. More precisely, the authors present an algorithm that solves the rendezvous problem for any n-node graph G, for any labels $L_1 > L_2 = l$ of agents and for any delay τ between startup times, in cost $\mathcal{O}(n^5 \sqrt{\tau \log l} \log n + n^{10} \log^2 n \log l)$. The algorithm contains a non-constructive component: agents use combinatorial objects whose existence is proved by the probabilistic method. Nevertheless the algorithm is indeed deterministic. Both agents can find separately the same combinatorial object with the desired properties (which is then used in the rendezvous algorithm). This can be done using brute force exhaustive search which may be quite complex but in the adopted model only moves of the agents are counted and computation time of the agents does not contribute to cost. Moreover, the authors note that finding this combinatorial object can be done only once at a preprocessing stage, the object can be stored in agents' memories and subsequently used in many instances of the rendezvous problem.

Finally, the authors prove a lower bound $\Omega(n^2)$ on the cost of rendezvous in some family of graphs. If simultaneous startup is assumed, they construct a simple generic rendezvous algorithm, working for all connected graphs, which is optimal for the class of graphs of bounded degree, if the initial distance between agents is bounded.

The paper is concluded by an open problem concerning the dependence of rendezvous cost on the delay τ. The dependence on the other parameters follows from the results cited above. Indeed, a lower bound $\Omega(n^2)$ on rendezvous cost has been shown in some graphs. The authors also showed that cost $\Omega(\log l)$ is required even for the two-node graph. On the other hand, for agents starting at distance $\Omega(n)$ in a ring, cost $\Omega(n \log l)$ is required, even for $\tau = 0$. However, the authors ask if any non-constant function of τ is a lower bound on rendezvous cost in some graphs. (Recall that the cost of their algorithm for arbitrary connected graphs contains a factor $\sqrt{\tau}$.) More precisely, they state the following problem:

Does there exist a deterministic rendezvous algorithm for arbitrary connected graphs with cost polynomial in n and l (or even in n and $\log l$) but independent of τ?

A positive answer to this problem has been given in [18] (whose conference version was published in 2006). Again, the authors restrict attention to the rendezvous of two agents, observing how this can be generalized for larger numbers of agents, assuming that agents meeting at the same node can exchange information. The authors present a rendezvous algorithm, working in arbitrary connected graphs for an arbitrary delay τ, whose complexity is $O(\log^3 l + n^{15} \log^{12} n)$, i.e., is independent of τ and polynomial in n and $\log l$. As before, the algorithm contains a non-constructive component, but is deterministic.

In the same paper, one of the results from [10], concerning rendezvous in the ring, is strengthened. Recall that the authors of [10] provided two rendezvous algorithms in a n-node ring with arbitrary delay: an algorithm with cost $O(n \log l)$, for known n, and an algorithm with cost $O(l\tau + ln^2)$, if n is unknown. By contrast, an algorithm with cost $O(n \log l)$ for *unknown* n is provided in [18]. In view of the lower bound $\Omega(n + D \log l)$ proved in [10], the latter cost complexity is optimal for some initial positions of the agents (when D is $\Theta(n)$).

The rendezvous algorithms from [10,18], working for arbitrary connected graphs, yield an intriguing question, stated in [10]. While both of them have polynomial cost (the one from [10] depending on τ, and the one from [18] independent of τ), they both use a non-constructive component, i.e, a combinatorial object whose existence is proved using the probabilistic method. As mentioned above, each of the agents can deterministically find such an object by exhaustive search, which keeps the algorithm deterministic, but may significantly increase the time of local computations. In the described model the time of these computations does not contribute to cost which is measured by the number of steps, regardless of the time taken to compute each step. Nevertheless, it is interesting if there exists a rendezvous algorithm for which both the cost and the time of local computations are polynomial in n and $\log l$. Such an algorithm would have to eliminate any non-constructive components. A positive answer to this question has been given in [22].

A different scenario and a different optimization goal have been used in [6,14]. In these papers it is assumed that agents are anonymous, i.e., they use the same algorithm without any parameter. In this case rendezvous is not possible for arbitrary networks and arbitrary initial positions of the agents, as witnessed by the example of the line of even size with symmetric port labelings, mentioned in Section 2. In order to describe initial positions of the agents for which rendezvous is possible, we need the notion of a *view* from a node of a graph, introduced in [23]. 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.

A pair (u, v) of distinct nodes is called *symmetric*, if $\mathcal{V}(u) = \mathcal{V}(v)$. Initial positions forming a symmetric pair of nodes are crucial when considering the feasibility of rendezvous in arbitrary graphs. Indeed it follows from [6] that rendezvous is feasible, if and only if the initial positions of the agents are not a symmetric pair. For the particular case of the class of trees, this is equivalent to the non-existence of a port-preserving automorphism of the tree that carries one initial position to the other.

The aim in [6,14] was not optimizing the cost but the memory size of the agents that seek rendezvous. In order to model agents with bounded memory, the formalism of input/output automata traveling in the graph is used.

Since in the currently considered scenario the agents are identical, we assume that agents are copies A and A' of the same abstract state machine $\mathcal{A}$, starting at two distinct nodes v_A and $v_{A'}$. We will refer to such identical machines as a *pair of agents*.

The optimization criterion is the size of the memory of the agents, measured by the number of states of the corresponding automaton, or equivalently by the number of bits on which these states are encoded. An automaton with K states requires $\Theta(\log K)$ bits of memory.

In [14] (based on conference papers [12,13]) the authors focus attention on optimizing memory size of identical agents that permits rendezvous in trees. They assume that the port labeling is decided by an adversary aiming at preventing two agents from meeting, or at allowing the agents to meet only after having consumed a lot of resources, e.g., memory space. This yields the following definition used in [14]. A pair of agents initially placed at nodes u and v of a tree T solves the rendezvous problem if, for any port labeling of T, both agents are eventually in the same node of the tree in the same round.

The following definition is crucial for considerations in [14]. Nodes u and v of a tree $T = (V, E)$ are *perfectly symmetrizable* if there exists a port labeling μ of T and an automorphism of the tree preserving μ that carries one node on the other. According to the above definition, the condition on feasibility of rendezvous can be reformulated as follows: a pair of agents can solve the rendezvous problem in a tree, if and only if their initial positions are not perfectly symmetrizable. Consequently, throughout [14], the authors consider only non perfectly symmetrizable initial positions of the agents.

It is first shown that the minimum size of memory of the agents that can solve the rendezvous problem in the class of trees with at most n nodes is $\Theta(\log n)$. A rendezvous algorithm for arbitrary delay τ, that uses only a logarithmic number of memory bits is a consequence, e.g., of [6]. It is observed in [14] that $\Omega(\log n)$ is also a lower bound on the number of bits of memory that enable rendezvous in all trees of size linear in n.

Due to the above lower bound, a *universal* pair of finite agents achieving rendezvous in the class of all trees cannot exist. However, the lower bound uses a counterexample of a tree with maximum degree linear in the size of the tree. Hence, it is natural to ask if there exists a pair of finite agents solving the rendezvous problem in all trees of *bounded* degree. The authors give a negative answer to this question. In fact they show that, for any pair of identical finite agents, there is a line on which these agents cannot solve the rendezvous problem, even with simultaneous start. As a function of the size of the trees, this impossibility result indicates a lower bound $\Omega(\log\log n)$ bits on the memory size for rendezvous in bounded degree trees of at most n nodes.

The main topic of [14] is the impact of the delay between startup times of agents on the minimum size of memory permitting rendezvous. The authors show that if this delay is arbitrary, then the lower bound on memory required for rendezvous is $\Omega(\log n)$ bits, even for the line of length n. This lower bound matches the upper bound from [6], which shows that the minimum size of memory of the agents that can solve the rendezvous problem in the class of *bounded degree* trees with at most n nodes is $\Theta(\log n)$. By contrast, for simultaneous start, they show that the amount of memory needed for rendezvous depends on two parameters of the tree: the number n of nodes and the number ℓ of leaves. Indeed, they show two identical agents with $O(\log\ell + \log\log n)$ bits of memory that solve the rendezvous problem in all trees with n nodes and ℓ leaves. For the class of trees with $O(\log n)$ leaves, this shows an exponential gap in minimum memory size needed for rendezvous between the scenario with arbitrary delay and with delay zero.

Moreover, it is shown in [14] that the size $O(\log\ell + \log\log n)$ of memory used to solve the rendezvous problem with simultaneous start in trees with at most n nodes and at most ℓ leaves is optimal, even in the class of trees with degrees bounded by 3. More precisely, for infinitely many integers ℓ, the authors show a class of arbitrarily large trees with maximum degree 3 and with ℓ leaves, for which rendezvous with simultaneous start requires $\Omega(\log\ell)$ bits of memory. This lower bound, together with the previously mentioned result showing that $\Omega(\log\log n)$ bits of memory are required for rendezvous with simultaneous start in the line of length n, implies that the upper bound $O(\log\ell + \log\log n)$ cannot be improved even for trees with maximum degree 3.

Trade-offs between the size of memory and the time of accomplishing rendezvous in trees by identical agents are investigated in [7]. The authors consider trees with a given port labeling and assume that there is no port-preserving automorphism of the tree that carries the initial position of one agent to that of the other (otherwise rendezvous with simultaneous start is impossible). The main result of the paper is a tight trade-off between optimal time of completing rendezvous and the size of memory of the agents. For agents with k memory bits, it is shown that optimal rendezvous time is $\Theta(n + n^2/k)$ in n-node trees. More precisely, if $k \geq c\log n$, for some constant c, the authors show agents accomplishing rendezvous in arbitrary trees of unknown size n in time $O(n + n^2/k)$, starting with arbitrary delay. They also show that no pair of agents can accomplish rendezvous in time $o(n + n^2/k)$, even in the class of lines and even with simultaneous start.

Trade-offs between the size of memory and the time of accomplishing rendezvous are investigated in [3] in a slightly different model. The authors consider rendezvous of any number of anonymous agents. To handle the case of symmetric trees they weaken the rendezvous requirements: agents have to meet at one node if the tree is not symmetric, and at two neighboring nodes otherwise. They observe that $\Omega(n)$ is a lower bound on the time of rendezvous in the class of n-node trees and show that any algorithm accomplishing rendezvous in optimal (i.e., linear) time must use $\Omega(n)$ bits of memory at each agent. Then they show a rendezvous algorithm that uses $O(n)$ time and $O(n)$ bits of memory per agent. Finally they show a polynomial time algorithm using $O(\log n)$ bits of memory per agent. An additional feature of the algorithms from [3] is that they can also work in an asynchronous setting: each agent independently identifies the node or one of the two nodes where meeting should occur, it reaches this node and stops.

While [14] solves the problem of minimum memory size needed for rendezvous in trees, the same problem for the class of arbitrary connected graphs is solved in [6]. The authors consider graphs with a given port labeling and establish the minimum size of the memory of agents that guarantees deterministic rendezvous when it is feasible. They show that this minimum size is $\Theta(\log n)$, where n is the size of the graph, regardless of the delay between the startup times of the agents. More precisely, the authors construct identical agents equipped with $\Theta(\log n)$ memory bits that solve the rendezvous problem in all graphs with at most n nodes, when starting with any delay τ, and they prove a matching lower bound $\Omega(\log n)$ on the number of memory bits needed to accomplish rendezvous, even for simultaneous start. In fact, this lower bound is achieved already on the class of rings.

4 Asynchronous Rendezvous

In asynchronous rendezvous agents no longer perform their moves in synchronized steps. While the agent chooses the adjacent node to which it wants to go, the time at which this move is executed is chosen by an adversary, which considerably complicates rendezvous. Two asynchronous models for rendezvous in networks have been used in the literature. The first is adapted from the CORDA model, originally used for rendezvous in the plane, cf. [5]. This model assumes that the agents are very weak in terms of memory (they cannot remember any past events), but they have significant sensory power (they can take snapshots of the entire network, including other agents' positions in it).

The model adapted from CORDA has been used in [16,17] to study rendezvous of many agents in a ring, in which neither nodes nor ports have labels. The authors concentrated on the rendezvous feasibility problem: for which initial configurations of agents rendezvous is possible? Agents are anonymous and execute the same algorithm. They start at different nodes and operate in Look-Compute-Move cycles. In one cycle, a robot takes a snapshot of the current configuration (Look), then, based on the perceived configuration, makes a decision to stay idle or to move to one of its adjacent nodes (Compute), and in the latter case makes an instantaneous move to this neighbor (Move). Cycles are performed asynchronously for each agent. This means that the time between Look, Compute, and Move operations is finite but unbounded, and is decided by the adversary for each agent in each cycle. The only constraint is that moves are instantaneous, and hence any agent performing a Look operation sees all other agents at nodes of the ring and not on edges, while performing a move. (This is where the model differs from the original CORDA model in which agents could be perceived during a move in the plane.) However, an agent A may perform a Look operation at some time t, perceiving agents at some nodes, then Compute a target neighbor at some time $t' > t$, and Move to this neighbor at some later time $t'' > t'$ in which some agents are in different nodes from those previously perceived by A because in the meantime they performed their Move operations. Agents are memoryless (oblivious), i.e., they do not have any memory of past observations. Thus the target node is decided by the agent during a Compute operation exclusively on the basis of the location of other agents perceived in the Look operation performed in the same cycle.

An important capability studied in the literature on rendezvous is the *multiplicity detection* [5]. This is the ability of the agents to perceive, during the Look operation, if there is one or more agents at a given node (or at a given point in the case of the plane). In the case of the ring, it is proved in [17] that without this capability, rendezvous of more than one agent is always impossible. Thus the authors assume the capability of multiplicity detection. Note that an agent can only tell if at some node there are no agents, there is one agent, or there are more than one agents, but it cannot determine the number of agents at a node.

The main result of [17] is the solution of the rendezvous problem for all initial configurations of any odd number of agents, for which rendezvous is possible. The authors prove that, for an odd number of agents, rendezvous is feasible, if and only if the initial configuration is not periodic, and they provide a rendezvous algorithm for any such configuration. (A configuration is periodic, if it is fixed by some non-trivial rotation of the ring, in the sense that such a rotation moves occupied nodes to occupied nodes and empty nodes to empty nodes.)

For an even number of agents, it is proved in [17] that rendezvous is impossible, if either the number of agents is 2, or the initial configuration is periodic, or when it has a symmetry axis on which there are no nodes. On the other hand, the authors provide a rendezvous algorithm for all rigid initial configurations, i.e., such in which all views of robots are distinct. This leaves unsettled one type of configurations: symmetric non-periodic configurations of an even number of agents with a node-on-axis type of symmetry. These are symmetric non-periodic configurations in which at least one node is situated on the unique axis of symmetry. It is conjectured in [17] that in the unique case left open (non-periodic configurations of an even number of agents with a node-on-axis symmetry), rendezvous is always feasible. This conjecture has been confirmed in [16], for all initial configurations of more than 18 agents. The authors provide a rendezvous algorithm for all such symmetric non-periodic configurations of an even number of agents with a node-on-axis type of symmetry.

A slightly different scenario was considered in [15]. The authors assume that agents have only the *local multiplicity detection* capability: every agent can only recognize whether the node in which it is currently situated contains another agent or not. Agents are unable to tell if another node is occupied by one or by more agents. Under this weaker assumption the authors consider rendezvous of an arbitrary number $2 < k \leq \lfloor n/2 \rfloor - 1$ of agents in a n-node ring, starting from a rigid initial configuration. For such configurations they show an algorithm accomplishing rendezvous in $O(n)$ asynchronous rounds. (Such a round is defined as the shortest fragment of an execution in which each agent is activated at least once.) This compares favorably to $O(kn)$ asynchronous rounds in [17].

A significantly different model of asynchrony for rendezvous has been used in [8,9,21]. Two agents are considered, with rendezvous occurring either at a node or inside an edge. Agents have distinct labels. Each of them knows its own label, but not that of the other agent. Nodes are anonymous and ports at each node are labeled in a possibly incoherent way. Since meetings inside an edge are allowed, unwanted crossings of edges have to be avoided. Thus, the authors consider an embedding of the underlying graph in the three-dimensional Euclidean space, with nodes of the graph being points of the space and edges being pairwise disjoint line segments joining them. For any graph such an embedding exists. Agents are modeled as points moving inside this embedding.

An algorithm for agent with label L depends on L and causes the agent to make the following decision at any node of the graph: either take a specific already explored incident edge, or take a yet unexplored incident edge (in which case the choice of the edge is made by the adversary, to model the worst case, since it is not assumed in general that agents know the topology of the graph).

There is another choice given to the adversary, this one capturing the asynchronous characteristics of the rendezvous process. When the agent, situated at a node v has to traverse an edge modeled as a segment $[v, w]$, the adversary chooses the actual movement of the agent on this segment, which can be at arbitrary, possibly varying speed, and it chooses the starting time of the agent.

For a given algorithm, given starting nodes of agents and a given sequence of adversarial decisions in an embedding of a graph G, a rendezvous occurs, if both agents are at the same point of the embedding at the same time. Rendezvous is *feasible* in a given graph, if there exists an algorithm for agents such that for any embedding of the graph, any (adversarial) choice of two distinct labels of agents, any starting nodes and any sequences of adversarial decisions, the rendezvous does occur. The *cost* of rendezvous is defined as the worst-case number of edge traversals by both agents (the last partial traversal counted as a complete one for both agents), where the worst case is taken over all decisions of the adversary.

In [9] the authors concentrate on minimizing the cost of rendezvous and they study three cases. First they consider rendezvous in an infinite line. For agents initially situated at a distance D in an infinite line, they show a rendezvous algorithm with cost $O(D|L_{min}|^2)$ when D is known, and $O((D + |L_{max}|)^3)$ if D is unknown, where $|L_{min}|$ and $|L_{max}|$ are the lengths of the shorter and longer label of the agents, respectively. These results still hold for the case of the ring (even of unknown size) but then an algorithm of cost $O(n|L_{min}|)$ is also given (and this is optimal), if the size n of the ring is known, and of cost $O(n|L_{max}|)$, if it is unknown. In both these algorithms the knowledge of the initial distance D between agents is not assumed, and for D of the order of n their complexity is better than that of infinite line algorithms. On the other hand, for small D and small labels of agents, the opposite is true.

For arbitrary graphs, it is proved that rendezvous is feasible, if an upper bound on the size of the graph is known, and an optimal algorithm of cost $O(D|L_{min}|)$ is given, if the topology of the graph and the initial positions are known to the agents. As an open problem, the authors state the question if asynchronous deterministic rendezvous is feasible in arbitrary graphs of unknown size. The solution from [9] uses the knowledge of the upper bound on the size.

In [21] the results from [9] for the infinite line have been improved. For known D, the author proposes an improvement by a constant factor, while for the case of unknown D an algorithm with cost $O(D \log^2 D + D \log D|L_{max}| + D|L_{min}|^2 + |L_{max}||L_{min}| \log |L_{min}|)$ is given.

Finally, in [8] the problem of feasibility of asynchronous rendezvous for arbitrary graphs is solved. The authors propose an algorithm that accomplishes asynchronous rendezvous in any connected countable (finite or infinite) graph, for arbitrary starting nodes. A consequence of this result is a strong positive answer to the above problem from [9]: not only is rendezvous always possible, without the knowledge of any upper bound on the size of a finite (connected) graph, but it is also possible for all infinite (countable and connected) graphs.

The cost of the algorithm from [8] is at least exponential in the labels of the agents and the size of the graph, for the case of finite graphs. Thus a natural question is the following: *Does there exist a deterministic asynchronous rendezvous algorithm, working for all connected finite unknown graphs, with cost polynomial in the labels of the agents and in the size of the graph?*

A partial solution to this problem has been given in [4], for particular graphs and under strong additional assumptions. The authors consider rendezvous in an infinite two-dimensional grid, where ports are consistently labeled N, E, S, W, agents have correct compasses and know their initial coordinates in the grid, with respect to a common system of coordinates. They show an asynchronous rendezvous algorithm for δ-dimensional infinite grids with cost $O(d^\delta polylog(d))$, where d is the initial distance between the agents. This complexity is close to optimal, as $\Omega(d^\delta)$ is a lower bound on the cost of any asynchronous rendezvous algorithm in this setting.

References

1. Alpern, S.: Rendezvous search: A personal perspective. Operations Research 50, 772–795 (2002)
2. Alpern, S., Gal, S.: The theory of search games and rendezvous. International Series in Operations Research and Management Science. Kluwer Acad. Publ., Dordrecht (2003)
3. Baba, D., Izumi, T., Ooshita, F., Kakugawa, H., Masuzawa, T.: Space-optimal rendezvous of mobile agents in asynchronous trees. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 86–100. Springer, Heidelberg (2010)
4. Bampas, E., Czyzowicz, J., Gasieniec, L., Ilcinkas, D., Labourel, A.: Almost optimal asynchronous rendezvous in infinite multidimensional grids. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 297–311. Springer, Heidelberg (2010)

5. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Solving the Robots Gathering Problem. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1181–1196. Springer, Heidelberg (2003)
6. Czyzowicz, J., Kosowski, A., Pelc, A.: How to meet when you forget: Log-space rendezvous in arbitrary graphs. In: Proc. 29th Annual ACM Symposium on Principles of Distributed Computing (PODC 2010), pp. 450–459 (2010)
7. Czyzowicz, J., Kosowski, A., Pelc, A.: Time vs. space trade-offs for rendezvous in trees (submitted)
8. Czyzowicz, J., Labourel, A., Pelc, A.: How to meet asynchronously (almost) everywhere. In: Proc. 21st ACM-SIAM Symp. on Discrete Algorithms (SODA 2010), pp. 22–30 (2010)
9. 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)
10. Dessmark, A., Fraigniaud, P., Kowalski, D., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46, 69–96 (2006)
11. Flocchini, P., Mans, B., Santoro, N.: Sense of direction: definition, properties and classes. Networks 32, 165–180 (1998)
12. 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)
13. Fraigniaud, P., Pelc, A.: Delays induce an exponential memory gap for rendezvous in trees. In: Proc. 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2010), pp. 224–232 (2010)
14. Fraigniaud, P., Pelc, A.: Delays induce an exponential memory gap for rendezvous in trees, arXiv:1102.0467v1 [cs.DC]
15. Izumi, T., Izumi, T., Kamei, S., Ooshita, F.: Mobile robots gathering algorithm with local weak multiplicity in rings. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 101–113. Springer, Heidelberg (2010)
16. 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)
17. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theoretical Computer Science 390, 27–39 (2008)
18. Kowalski, D., Malinowski, A.: How to meet in anonymous network. Theoretical Computer Science 399, 141–156 (2008)
19. Kranakis, E., Krizanc, D., Markou, E.: The mobile agent rendezvous problem in the ring. Morgan and Claypool Publishers (2010)
20. Kranakis, E., Krizanc, D., Rajsbaum, S.: Mobile agent rendezvous: A survey. In: Flocchini, P., Gąsieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 1–9. Springer, Heidelberg (2006)
21. Stachowiak, G.: Asynchronous Deterministic Rendezvous on the Line. In: Nielsen, M., Kučera, A., Miltersen, P.B., Palamidessi, C., Tůma, P., Valencia, F. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 497–508. Springer, Heidelberg (2009)
22. 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)
23. Yamashita, M., Kameda, T.: Computing on Anonymous Networks: Part I-Characterizing the Solvable Cases. IEEE Trans. Parallel Distrib. Syst. 7, 69–89 (1996)

Fast and Scalable Rendezvousing

Yehuda Afek, Michael Hakimi, and Adam Morrison

School of Computer Science
Tel Aviv University

Abstract. In an asymmetric rendezvous system, such as an unfair synchronous queue and an elimination array, threads of two types, consumers and producers, show up and are matched, each with a unique thread of the other type. Here we present a new highly scalable, high throughput asymmetric rendezvous system that outperforms prior synchronous queue and elimination array implementations under both symmetric and asymmetric workloads (more operations of one type than the other). Consequently, we also present a highly scalable elimination-based stack.

1 Introduction

A common abstraction in concurrent programming is that of an *asymmetric rendezvous* mechanism. In this mechanism, there are two types of threads that show up, e.g., producers and consumers. The goal is to match pairs of threads, one of each type, and send them away. Usually the purpose of the pairing is for a producer to hand over a data item (such as a task to perform) to a consumer. The asymmetric rendezvous abstraction encompasses both *unfair synchronous queues* (or *pools*) [12] which are a key building block in Java's thread pool implementation and other message-passing and hand-off designs [2, 12], and the *elimination* technique [13], which is used to scale concurrent stacks and queues [7, 11].

In this paper we present a highly scalable asymmetric rendezvous algorithm that improves the state of the art in both unfair synchronous queue and elimination algorithms. It is based on a distributed scalable ring structure, unlike Java's synchronous queue which relies on a non-scalable centralized structure. It is *nonblocking*, in the following sense: if both producers and consumers keep taking steps, *some* rendezvous operation is guaranteed to complete. (This is similar to the *lock-freedom* property [8], while taking into account the fact that "it takes two to tango", i.e., both types of threads must take steps to successfully rendezvous.) It is also uniform, in that no thread has to perform work on behalf of other threads. This is in contrast to the flat combining (FC) based synchronous queues of Hendler et al. [6], which are blocking and non-uniform.

Our algorithm is based on a simple and remarkably effective idea: the *algorithm itself is asymmetric.* A consumer captures a node in the ring and waits there for a producer, while a producer actively seeks out waiting consumers on the ring. The algorithm utilizes a new *ring adaptivity scheme* that dynamically adjusts the ring size, leaving enough room for all the consumers while avoiding empty nodes that producers will futileness search. Because of the adaptive ring size, we can expect the nodes to be densely populated, and thus a producer that starts to scan the ring and finds a node to be empty, it is likely that a consumer will arrive there shortly. Yet simply waiting at this node, hoping that this will occur, makes the algorithm prone to timeouts and impedes progress. We solve this problem by introducing *peeking* a technique that lets the producer enjoy the best of both worlds: as the producer traverses the ring, it continues to peek at its initial node; if a consumer arrives there, the producer immediately tries to partner with it, thereby minimizing the amount of wasted work.

Our algorithm avoids two problems found in prior elimination algorithms that did not exploit asymmetry. In these works [1, 7, 13], both types of threads would pick a random node in the hope of meeting the right kind of partner. Thus these works suffer from *false matches*, when two threads of the same type meet, and from *timeouts*, when a producer and a consumer both pick distinct nodes and futilely wait for a partner to arrive. Most importantly, our algorithm performs extremely well in practice. On an UltraSPARC T2 Plus multicore machine, it outperforms Java's synchronous queue by up to 60×, the FC synchronous queue by up to 6×, and, when used as the elimination layer of a concurrent stack, yields 3.5× improvement over Hendler et al.'s FC stack [5]. On an Intel Nehalem multicore (supporting less parallelism), our algorithm surpasses the Java pool and the FC pool by 5× and 2×, respectively.

The asymmetric rendezvous problem and our progress property are formally defined in Sect. 2. Section 3 describes related work. The algorithm is presented in Sect. 4 and empirically evaluated in Sect. 5. We conclude in Sect. 6.

2 Preliminaries

Asymmetric rendezvous: In the *asymmetric rendezvous* problem there are threads of two types, producers and consumers. Producers perform `put(`x`)` operations that return `OK`. Consumers perform `get()` operations that return some item x handed off by a producer. Producers and consumers show up and must be matched with a unique thread of the other type, such that a producer invoking `put(`x`)` and a consumer whose `get()` returns x must be active concurrently.

Progress: To reason about progress while taking into account that rendezvous inherently requires waiting, we consider both types of operations' combined behavior. An algorithm **A** that implements asymmetric rendezvous is *nonblocking* if *some* operation completes after enough *concurrent* steps of threads performing `put()` operations *and* of threads performing `get()`. Note that, as with the definition of the lock-freedom property [8], there is no fixed a priori bound on the number of steps after which some operation must complete. Rather, we rule out implementations that make no progress at all, i.e., implementations where in some executions, both types of threads take steps infinitely often and yet no operation completes.

3 Related Work

Synchronous queues: A synchronous queue using three semaphores was described by Hanson [4]. Java 5 includes a coarse-grained locking synchronous queue, which was superseded in Java 6 by Scherer, Lea and Scott's algorithm [12]. Their algorithm is based on a Treiber-style nonblocking stack [16] that at all times contains rendezvous requests by either producers or consumers. A producer finding the stack empty or containing producers pushes itself on the stack and waits, but if it finds the stack holding consumers, it attempts to partner with the consumer at the top of the stack (consumers behave symmetrically). This creates a sequential bottleneck. Motivated by this, Afek, Korland, Natanzon, and Shavit described *elimination-diffracting (ED) trees* [1], a randomized distributed data structure where arriving threads follow a path through a binary tree whose internal nodes are *balancer* objects [14] and the leaves are Java synchronous queues. In each internal node a thread accesses an elimination array in attempt to avoid descending down the tree. Recently, Hendler et al. applied the *flat combining* paradigm [5] to the synchronous queue problem [6], describing single-combiner and parallel versions. In a single combiner FC pool, a thread attempts to become a *combiner* by acquiring a global lock on the queue. Threads that fail to grab the lock instead post their request and wait for it to be fulfilled by the combiner, which matches between the participating threads. In the parallel version there are multiple combiners that each handle a subset of participating threads and then try to satisfy unmatched requests in their subset by entering an *exchange* FC synchronous queue.

Elimination: The elimination technique is due to Touitou and Shavit [13]. Hendler, Shavit and Yerushalmi used elimination with an adaptive scheme inspired by Shavit and Zemach [15] to obtain a scalable linearizable stack [7]. In their scheme threads adapt locally: each thread picks a slot to collide in from sub-range of the collision layer centered around the middle of the array. If no partner arrives, the thread eventually shrinks the range. Alternatively, if the thread sees a waiting partner but fails to collide due to contention, it increases the range. In our adaptivity technique, described in Sect. 4, threads also make local decisions, but with global impact: the ring is resized. Moir et al. used elimination to scale a FIFO queue [11]. In their algorithm an enqueuer picks a random slot in an elimination array and waits there for a dequeuer; a dequeuer picks a random slot, giving up immediately if that slot is empty. It does not seek out waiting enqueuers. Scherer, Lea and Scott applied elimination in their *symmetric* rendezvous system [9], where there is only one type of a thread and so the pairing is between any two threads that show up. Scherer, Lea and Scott also do not discuss adaptivity, though a later version of their symmetric *exchanger channel*, which is part of Java [10], includes a scheme that resizes the elimination array. However, here we are interested in the more difficult *asymmetric* rendezvous problem, where not all pairings are allowed.

4 Algorithm Description

The pseudo code of the algorithm is provided in Fig. 1. The main data structure (Fig. 1a) is a ring of nodes. The ring is accessed through a central array `ring`, where `ring[`i`]` points to the i-th node in the ring.[1] A consumer attempts to capture a node in the ring and waits there for a producer, whereas a producer scans the ring, seeking a waiting consumer. Therefore, to guarantee progress the ring must have room for all the consumers that can be active simultaneously. (We expand on this in Sect. 4.4.) For simplicity, we achieve this by assuming the number of threads in the system, T, is known in advance and pre-allocating a ring of size T. In Sect. 4.3 we sketch a variant in which the maximum number of threads that can show up is not known in advance, i.e., adaptive also to the number of threads.

Conceptually each ring node should only contain an `item` pointer that encodes the node's state:

1. **Free**: `item` points to a global reserved object, `FREE`, that is distinct from any object a producer may enqueue. Initially all nodes are free.
2. **Captured by consumer:** `item` is `NULL`.
3. **Holding data (of a producer):** `item` points to the data.

In practice, ring traversal is more efficient by following a pointer from one node to the next rather than the alternative, traversing the array. Array traversal suffers from two problems. First, in Java reading from the next array cell may result in a cache miss (it is an array of pointers), whereas reading from the (just accessed) current node does not. Second, maintaining a running array index requires an expensive test+branch to handle index boundary conditions or counting modulo the ring size, while reading a pointer is cheap. The pointer field is named `prev`, reflecting that node i points to node $i-1$. This allows the ring to be *resized* with a single atomic `compareAndSet` (CAS) that changes `ring[1]`'s (the head's) `prev` pointer. To support mapping from a node to its ring index, each node holds its ring index in a read-only `index` field.

For the sake of clarity we start in Sect. 4.1 by discussing the algorithm without adaptation of the ring size. The adaptivity code, however, is included and is marked by a $\triangleright$ symbol in Fig. 1, and is discussed in Sect. 4.2. Section 4.3 sketches some practically-motivated extensions to the algorithm, e.g. supporting timeouts and adaptivity to the total number of threads. Finally, Sect. 4.4 discusses correctness and progress.

4.1 Nonadaptive Algorithm

Producers (Fig. 1b): A producer searches the ring for a waiting consumer, and attempts to hand its data to it. The search begins at a node, s, obtained by hashing the thread's id (Line 19). The producer passes the ring size to the hash

[1] This reflects Java semantics, where arrays are of *references* to objects and not of objects themselves.

function as a parameter, to ensure the returned node falls within the ring. It then traverses the ring looking for a node captured by a consumer. Here the producer periodically *peeks* at the initial node s to see if it has a waiting consumer (Lines 24-26); if not, it checks the current node in the traversal (Lines 27-30). Once a captured node is found, the producer tries to deposit its data using a CAS (Lines 25 and 28). If successful, it returns.

```
 1 struct node {
 2   item : pointer to object
 3   index : node's index in the ring
 4   prev : pointer to previous ring node
 5 }
 6
 7 shared vars:
 8   ring : array [1,...,T] of pointers to nodes,
 9          ring [1]. prev = ring[T],
10          ring[ i ]. prev = ring[i−1] (i > 1)
11
12 utils:
13 getRingSize() {
14   node tail := ring [1]. prev
15   return tail.index
16 }
```

(a) Global variables

```
17 put(threadId, object item) {
18
19   node s := ring[hash(threadId,
20                       getRingSize())]
21   node v := s.prev
22
23   while (true) {
24     if (s.item == NULL) {
25       if (CAS(s.item, NULL, item))
26         return OK
27     } else if (v.item == NULL) {
28       if (CAS(v.item, NULL, item))
29         return OK
30       v := v.prev
31     }
32   }
33 }
```

(b) Producer code

```
34 get(threadId) {
35   int ringsize , busy_ctr, ctr := 0
36   node s, u
37
38   ringsize := getRingSize()
39   s := ring[hash(threadId, ringsize )]
40   (u,busy_ctr) := findFreeNode(s,ringsize)
41   while (u.item == NULL) {
42 ▷   ringsize := getRingSize()
43 ▷   if (u.index > ringsize and
44 ▷       CAS(u.item, NULL, FREE) {
45 ▷     s := ring[hash(threadId, ringsize )]
46 ▷     (u,busy_ctr) := findFreeNode(s, ringsize )
47 ▷   }
48 ▷   ctr := ctr + 1
49   }
50
51   item := u.item
52   u.item := FREE
53 ▷ if (busy_ctr < $T_d$ and ctr > $T_w$ and ringsize>1)
54 ▷   // Try to decrease ring
55 ▷   CAS(ring[1].prev, ring[ ringsize ], ring[ ringsize −1])
56   return item
57 }
58
59 findFreeNode(node s, int ringsize ) {
60 ▷ int busy_ctr := 0
61   while (true) {
62     if (s.item == FREE and
63         CAS(s.item, FREE, NULL))
64       return (s,busy_ctr)
65     s := s.prev
66 ▷   busy_ctr := busy_ctr + 1
67 ▷   if (busy_ctr > $T_i$·ringsize) {
68 ▷     if ( ringsize < T) // Try to increase ring
69 ▷       CAS(ring[1].prev, ring[ ringsize ], ring[ ringsize +1])
70 ▷     ringsize := getRingSize()
71 ▷     busy_ctr := 0
72 ▷   }
73   }
74 }
```

(c) Consumer code

Fig. 1. Asymmetric rendezvous algorithm. Lines beginning with ▷ handle the adaptivity process.

Consumers (Fig. 1c): A consumer searches the ring for a free node and attempts to capture it by atomically changing its `item` pointer from `FREE` to `NULL` using a CAS. Once a node is captured, the consumer spins, waiting for a producer to arrive and deposit its item. Similarly to the producer, a consumer hashes its id to obtain a starting point for its search, s (Line 39). The consumer calls the `findFreeNode` procedure to traverse the ring from s until it captures and returns node u (Lines 59-74). (Recall that the code responsible for handling adaptivity, which is marked by a ▷, is ignored for the moment.) The consumer then waits until a producer deposits an item in u (Lines 41-49), frees u (Lines 51-52) and returns (Line 56).

4.2 Adding Adaptivity

If the number of active consumers is smaller than the ring size, producers may need to traverse through a large number (linear in the ring size) of empty nodes before finding a match. It is therefore important to *decrease* the ring size if the concurrency level is low. On the other hand, if there are more concurrent threads than the ring size (high contention), it is important to dynamically *increase* the ring. The goal of the adaptivity scheme is to keep the ring size "just right" and allow threads to complete their operations within a constant number of steps.

The logic driving the resizing process is in the consumer's code, which detects when the ring is overcrowded or sparsely populated and changes the size accordingly. If a consumer fails to capture many nodes in `findFreeNode()` (due to not finding a free node or having its CASes fail), then the ring is too crowded and should be increased. The exact threshold is determined by an *increase threshold* parameter, T_i. If `findFreeNode()` fails to capture more than $T_i \cdot$ `ring_size` it attempts to increase the ring size (Lines 67-72). To detect when to decrease the ring, we observe that when the ring is sparsely populated, a consumer usually finds a free node quickly, but then has to wait longer until a producer *finds it*. Thus, we add a *wait threshold*, T_w, and a *decrease threshold*, T_d. If it takes a consumer more than T_w iterations of the loop in Lines 41-49 to successfully complete the rendezvous, but it successfully captured its ring node in up to T_d steps, then it attempts to decrease the ring size (Lines 53-55).

Resizing the ring is made by CASing the `prev` pointer of the ring head (`ring[1]`) from the current tail of the ring to the tail's successor (to increase the ring size) or its predecessor (to decrease the ring size). If the CAS fails, then another thread has resized the ring and the consumer continues. The head's `prev` pointer is not a sequential bottleneck because resizing is a rare event in stable workloads. Even if resizing is frequent, the thresholds ensure that the cost of the CAS is negligible compared to the other work performed by the algorithm, and resizing pays-off in terms of better ring size which leads to improved performance.

Handling consumers left out of the ring: A decrease in the ring size may leave a consumer's captured node outside of the current ring. Therefore, each consumer periodically checks if its node's index is larger than the current ring size (Line 43). If so, it tries to free its node using a CAS (Line 44) and find itself a new node in the ring (Lines 45-46). However, if the CAS fails, then a producer has already deposited its data in this node and so the consumer can take the data and return (this will be detected in the next execution of Line 41).

4.3 Pragmatic Extensions and Considerations

Here we sketch a number of extensions that might be interesting in certain practical scenarios.

Timeout support: Aborting a rendezvous that is taking more than a specified period of time is a useful feature. Unfortunately, our model has no notion of time and so we do not model the semantics of timeouts. We simply describe the details for an implementation that captures the intuitive notion of a timeout. The operations take a parameter specifying the desired timeout, if any. Timeout expiry is then checked in each iteration of the main loop in `put()`, `get()` and `findFreeNode()`. If the timeout expires, a producer or a consumer in findFreeNode() aborts by returning. A consumer that has captured a ring node cannot abort before freeing the node by CASing its `item` from `NULL` back to `FREE`. If the CAS fails, the consumer has found a match and its rendezvous has completed successfully. Otherwise its abort attempt succeeded and it returns.

Avoiding ring pre-allocation: Ring pre-allocation requires in advance knowledge of the maximum number of threads that can simultaneously run, which may not be known a priori. The maximum ring size can be dynamic by allowing threads to add and delete nodes from the ring. To do this we exploit the Java semantics, which forces the `ring` variable to be a *pointer* to the array. This allows a consumer to allocate a new ring (larger or smaller) and CAS `ring` to point to it. Active threads periodically check if `ring` has changed and move themselves to the new ring. We omit the details due to space limitations.

Busy waiting: Both producers and consumers rely on busy waiting, which may not be appropriate for *blocking* applications that wish to let a thread sleep until a partner arrives. Being blocking, such applications may not need our strong progress guarantee. We plan to extend our algorithm to blocking scenarios in future work.

4.4 Correctness

We consider the point at which both `put(`x`)` and the `get()` that returns x take effect as the point where the producer successfully CASes x into some node's `item` pointer (Line 25 or 28). The algorithm thus clearly meets the asymmetric rendezvous semantics. We next sketch a proof that, assuming threads do not request timeouts, the algorithm is nonblocking (as defined in Sect. 2). Assume towards a contradiction that there is an execution in which both producers and consumers take infinitely many steps and yet no rendezvous successfully completes. Since there is a finite number of threads T, there must be a producer/-consumer pair, p and c, each of which runs forever without completing. Suppose c never captures a node. Then eventually the ring size must become T. This is because after c completes a cycle around the ring in `findFreeNode()`, it tries to increase the ring size (say the increase threshold is 1). c's resizing CAS (Line 69) either succeeds or fails due to another CAS that succeeded in increasing, because the ring size decreasing implies a rendezvous completing, which by our assumption does not occur. Once the ring size reaches T, the ring has room for c (since T is the maximum number of threads). Thus c fails to capture a node only by encountering another consumer twice at different nodes, implying this consumer completes a rendezvous, a contradiction. It therefore cannot be that c never captures a node. Instead, c captures some node but p never finds c on the ring, implying that c has been left out of the ring by some decreasing resize. But, since from that point on, the ring size cannot decrease, c eventually executes Lines 43-46 and moves itself into p's range, where either p or another producer rendezvous with it, a contradiction.

5 Evaluation

We evaluate the performance of our algorithm in comparison with prior unfair synchronous queues (Sect. 5.1) and demonstrate the performance gains due to the adaptivity and the peeking techniques). Then we evaluate the resulting stack performance against other concurrent stacks (Sect. 5.2).

Experimental setup: Evaluation is carried out on both a Sun SPARC T5240 and on an Intel Core i7. The Sun has two UltraSPARC T2 Plus (Niagara II) chips. Each is a multithreading (CMT) processor, with 8 1.165 HZ in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip. The Intel Core i7 920 (Nehalem) processor has four 2.67GHz cores, each multiplexing 2 hardware threads. Both Java and C++ implementations are tested.[2] Due to space limitations, we focus more on the Sun platform which offers more parallelism and better insight into the scaling behavior of the algorithm. Unless stated otherwise, results are averages of ten 10-second runs of the Java implementation on an idle machine, resulting in very little variance.

Adaptivity parameters used: $T_i = 1$, $T_d = 2$, and $T_w = 64$. A modulo hash function, $\texttt{hash}(t) = t \mod \texttt{ringsize}$, is used since we don't know anything about the thread ids and consider them as uniformly random. If however thread ids are *sequential* the modulo hash achieves perfect coverage of the ring with few collisions and helps in pairing producers/consumers that often match with each other. We evaluate these effects by testing our algorithms with both random and sequential thread ids throughout all experiments.

5.1 Synchronous Queues

Our algorithm (marked **AdaptiveAR** in the figures) is compared against the Java unfair synchronous queue (JDK), ED tree, and FC synchronous queues. The original authors' implementation of each algorithm is used.[3] We tested both JDK

[2] Java benchmarks were ran with HotSpot Server JVM, build `1.7.0-ea-b137`. C++ benchmarks were compiled with Sun C++ 5.9 on the SPARC machine and with `gcc` 4.3.3 (`-O3` optimization setting) on the Intel machine. In the C++ experiments we used the Hoard 3.8 [3] memory allocator.

[3] We remove all statistics counting from the code and use the latest JVM. Thus, the results we report are usually slightly better than those reported in the original papers. On the other hand, we fixed a bug in the benchmark of [6] that miscounted timed-out operations of the Java pool as successful operations; thus the results we report for it are sometimes lower.

and JDK-no-park: a version that always uses busy waiting instead of yielding the CPU (so-called parking), and report its results for the workloads where it improves upon the standard Java pool (as was done in [6]).

Producer/Consumer Throughput

$N:N$ producer/consumer symmetric workload: We measure the throughput at which data is transferred from N producers to N consumers. We focus first on single chip results in Figs. 2a and 2e. The Nehalem results are qualitatively similar to the low thread counts SPARC results. Other than our algorithm, the parallel FC pool is the only algorithm that shows meaningful scalability. Our rendezvous algorithm outperforms the parallel FC queue by $2\times - 3\times$ in low concurrency settings, and by up to 6 at high thread counts.

Hardware performance counter analysis (Fig. 2b-2d) shows that our rendezvous completes in less than 170 instructions, of which one is a CAS. In the parallel FC pool, waiting for the combiner to pick up a thread's request, match it, and report back with the result, all add up. While the parallel FC pool hardly

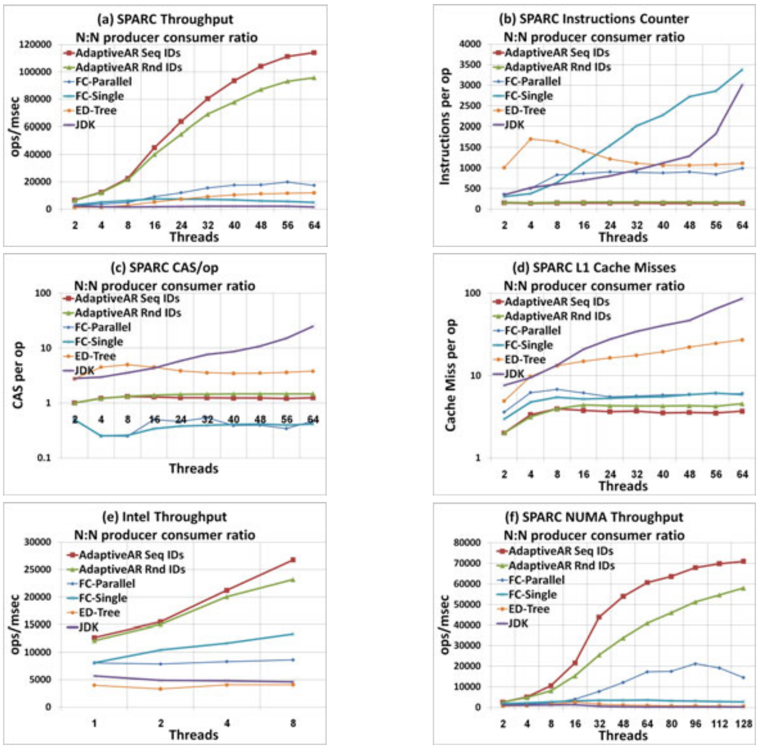


Fig. 2. Rendezvousing between N pairs of producers and consumers. Performance counter plots are logarithmic scale. L2 misses are not shown; all algorithms but JDK had less than one L2 miss/operation on average.

performs CASes, it does require between 3× to 6× more instructions to complete an operation. Similarly, ED tree's rendezvous operations require 1000 instructions to complete and incur more cache misses as concurrency increases. Our algorithm therefore outperforms it by at least 6× and up to 10× at high thread counts. The Java synchronous queue fails to scale in this benchmark. Figures 2b and 2c show its serializing behavior. Due to the number of failed CAS operations on the top of the stack (and consequent retries), it requires more instructions to complete an operation as concurrency grows. Consequently, our algorithm outperforms it by more than 60×.

Adaptivity and peeking impact: From Fig. 2c we deduce that ring resizing operations are a rare event, leading to an average number of one CAS per operation. Thus resizing does not adversely impact performance here. Throughput of the algorithm with and without peeking is compared in Figure 4a. While peeking has little effect at low thread counts, it improves performance by as much as 47% once concurrency increases. Because, due to adaptivity, a producer's initial node being empty usually means that a consumer will arrive there shortly, and peeking increasing the chance of pairing with that consumer. Without it a producer has a higher chance of colliding with another producer and consequently spending more instructions (and CASes) to complete a rendezvous.

NUMA test: When utilizing both processors of the Sun machine the operating system's default scheduling policy is to place threads round-robin on both chips. Thus, the cross-chip overhead is noticeable even at low thread counts, as Fig. 2f shows. Since the threads no longer share a single L2 cache, they experience an increased number of L1 and L2 cache misses; each such miss is expensive, requiring coherency protocol traffic to the remote chip. The effect is catastrophic for serializing algorithms; for example, the Java pool experiences a 10× drop in throughput. The more concurrent algorithms, such as parallel FC and ours, show scaling trends similar to the single chip ones, but achieve lower throughput. In the rest of the section we therefore focus on the more interesting single chip case.

$1 : N$ *asymmetric workloads:* The throughput of one producer rendezvousing with a varying number of consumers is presented in Figs. 3a and 4c. Nehalem results (Fig. 4c) are again similar and not discussed in detail. Since the throughput is bounded by the rate of a single producer, little scaling is expected and observed. However, for all algorithms (but the ED tree) it takes several consumers to keep up with a single producer. This is because the producer hands off an item and completes, whereas the consumer needs to notice the rendezvous has occurred. And while the single consumer is thus busy the producer cannot make progress on a new rendezvous. However, when more consumers are active, the producer can immediately return and find another (different) consumer ready to rendezvous with. Unfortunately, as shown in Figure 3a, most algorithms do not sustain this peak throughput. The FC pools have the worst degradation (3.74× for the single version, and 3× for the parallel version). The Java pool's degradation is minimal (13%), and it along with the ED tree achieves close to

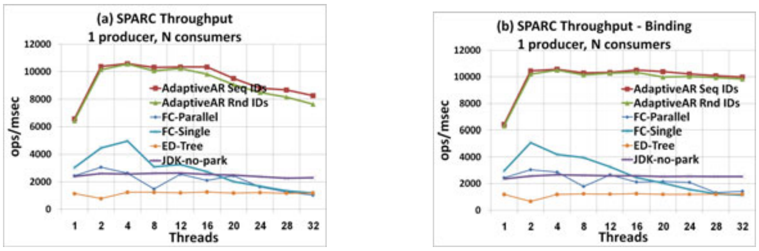


Fig. 3. Single producer and N consumers rendezvousing. **Left:** Default OS scheduling. **Right:** OS constrained to not co-locate producer on same cores as consumers.

peak throughput even at high thread counts. Yet this throughput is low: our algorithm outperforms the Java pool by up to 3× and the ED tree by 12× for low consumer counts and 6× for high consumer counts, despite degrading by 23% – 28% from peak throughput.

Why our algorithm degrades is not obvious. The producer has its pick of consumers in the ring and should be able to complete a hand-off immediately. The reason for this degradation is not algorithmic, but due to *contention on chip resources*. A Niagara II core has two pipelines, each shared by four hardware strands. Thus, beyond 16 threads some threads must share a pipeline — and our algorithm indeed starts to degrade at 16 threads. To prove that this is the problem, we present in Fig. 3b runs where the producer runs on the first core and consumers are scheduled only on the remaining seven cores. While the

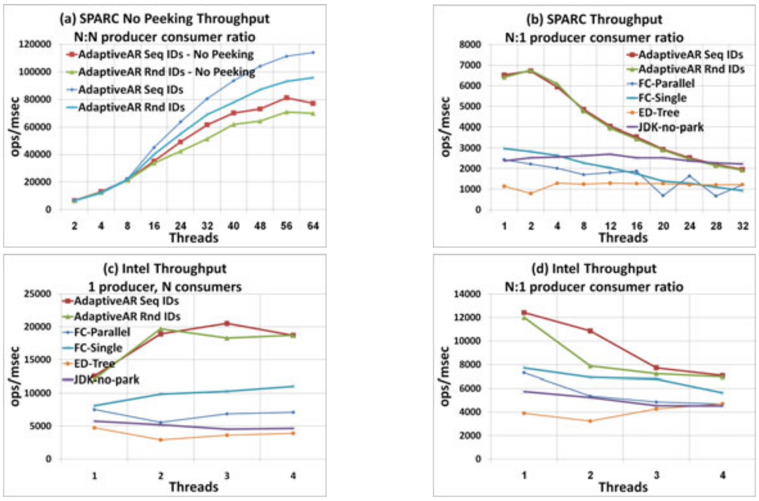


Fig. 4. Top left: Impact of peeking on $N : N$ rendezvous. **Top right:** Rendezvousing between N producers and a single consumer. **Bottom:** Intel $1 : N$ and $N : 1$ producer/consumer workloads throughput.

trends of the other algorithms are unaffected, our algorithm now maintains peak throughput through all consumer counts.

Results from the opposite workload (which is less interesting in real life scenarios), where multiple producers try to serve a single consumer, are given in Figs. 4b and 4d. Here the producers contend over the single consumer node and as a result the throughput of our algorithm degrades as the number of producers increases (as do the FC pools). Despite this degradation, our algorithm outperforms the Java pool up to 48 threads (falling behind by 15% at 64 threads) and outperforms the FC pools by about 2×.

Bursts: To evaluate the effectiveness of our adaptivity technique, we measure the rendezvous rate in a workload that experiences bursts of activity (on the Sun machine). For ten seconds the workload alternates every second between 31 thread pairs and 8 pairs. The 63rd hardware strand is used to take ring size measurement. Our sampling thread continuously reads the ring size, and records the time whenever the current read differs from the previous read. Figure 5a depicts the result, showing how the algorithm continuously resizes its ring. Consequently, it successfully benefits from the available parallelism in this workload, outperforming the Java pool by at least 40× and the parallel FC pool by 4× to 5×.

Varying arrival rate: In practice, threads do some work between rendezvouses. We measure how the throughput of the producer/consumer pairs workload is affected when the thread arrival rate decreases due to increasingly larger amounts

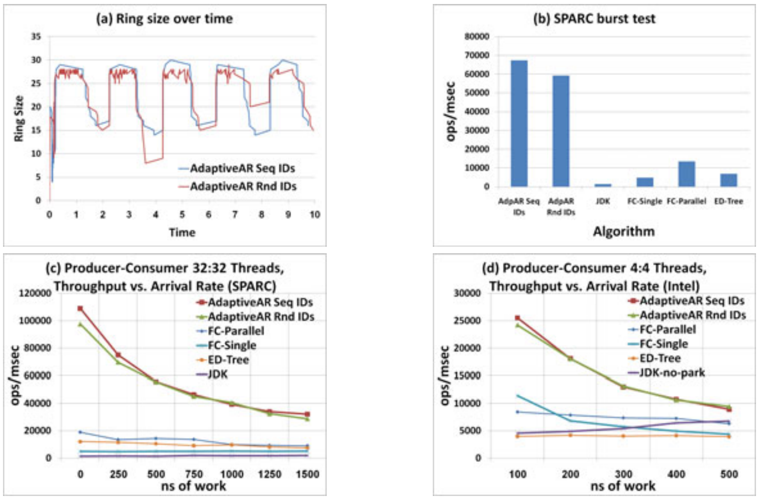


Fig. 5. Top: Synchronous queue bursty workload throughput. Left: Our algorithm's ring size over time (sampled continuously using a thread that does not participate in the rendezvousing). Right: throughput. **Bottom:** $N : N$ rendezvousing with decreasing arrival rate due to increasing amount of work time operations.

of time spent doing "work" before each rendezvous. Figures 5c and 5d show that as the work period grows the throughput of all algorithms that exhibit scaling deteriorates, due to reduced parallelism in the workload. E.g., on the SPARC the parallel FC degrades by 2× when going from no work to 1.5μs of work, and our algorithm degrades by 3.4× (because it starts much higher). Still, on the SPARC there is sufficient parallelism to allow our algorithm to outperform the other implementations by a factor of at least three. (On the Nehalem, by 31%.)

Work uniformity: One clear advantage of our algorithm over FC is work *uniformity*, since the combiner in FC is expected to spend much more time doing work for other threads. We show this by comparing the percent of total operations performed by each thread in a multiple producer/multiple consumer workload. In addition to measuring uniformity, this test is a yardstick for progress *in practice*: if a thread starves, we will see it as performing very little work compared to other threads. We pick the best result from five executions of 16 producer/consumer pairs, and plot the percent of total operations performed by each thread. Figure 6 shows the results. In an ideal setting, each thread would perform 3.125% (1/32) of the work. Our algorithm comes relatively close to this distribution, as does the JDK algorithm. In contrast, the parallel FC pool experiences much larger deviation and best/worst ratio.

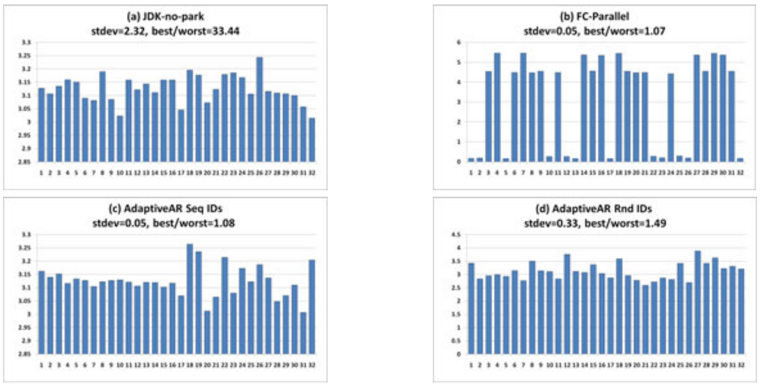


Fig. 6. Percent of total operations performed by each of 32 threads in an $N : N$ test

5.2 Concurrent Stack

Here we use our algorithm as the elimination layer on top of a Treiber-style nonblocking stack [16] and compare it to the stack implementations evaluated in [5]. We implemented two variants. In the first, following [7], rendezvous is used as a *backoff* mechanism that threads turn to upon detecting contention on the main stack, thus providing good performance under low contention and scaling as contention increases. In the second variant a thread first visits the rendezvous structure, accessing the main stack only if it fails to find a partner. If it then

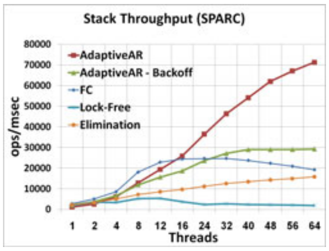


Fig. 7. Comparing stack implementations throughput. Each of N threads performs both `push` and `pop` operations with probability 1/2 for each operation type.

encounters contention on the main stack it goes back to try the rendezvous, and so on.

We compare a C++ implementation of our algorithm to the C++ implementations evaluated in [5]: a lock-free stack (LF-Stack), a lock-free stack with a simple elimination layer (EL-Stack), and an FC based stack. We use the same benchmark as [5], measuring the throughput of an even `push`/`pop` operation mix. Unlike the pool tests, here we want threads to give up if they don't find a partner in a short amount of time, and move to the main stack. We thus need to set T_w to a value smaller than the timeout, to enable a decrease of the ring size. Figure 7 shows the results. Our second (non-backoff) stack scales well, outperforming the FC and elimination stacks by more than 3.5×. The price it pays is poor performance at low concurrency (2.5× slower than the FC stack with a single thread). The backoff variant fails to scale above 32 threads due to contention on the stack head, illustrating the cost incurred by merely trying (and failing) to CAS a central hotspot.

6 Conclusion

We have presented an adaptive, nonblocking, high throughput asymmetric rendezvous system that scales well under symmetric workloads and maintains peak throughput in asymmetric (more consumers than producers) workloads. This is achieved by a careful marriage of new algorithmic ideas and attention to implementation details, to squeeze all available cycles out of the processors.

Several directions remain open towards making the algorithm even more broadly applicable and practical. In ongoing work we are extending the algorithm so that it maintains peak throughput even when there are more producers than consumers, and pursuing making the algorithm's space consumption adaptive. Adapting the algorithm to blocking applications (i.e., avoiding busy waiting) and dynamically choosing the various threshold parameters are also interesting questions.

Our results also raise a question about the flat combining paradigm. Flat combining has a clear advantage in inherently sequential data structures, such as a FIFO or priority queue, whose concurrent implementations have central hot spots. But as we have shown, FC may lose its advantage in problems with inherent potential for parallelism. It is therefore interesting whether the FC technique can be improved to match the performance of our asymmetric rendezvous system.

Availability: Our implementation is available on Tel Aviv University's Multicore Algorithmics group web site at `http://mcg.cs.tau.ac.il/`.

Acknowledgments: We thank Hillel Avni, Nati Linial and Nir Shavit for helpful discussions, and the anonymous reviewers for their comments.

References

[1] Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 151–162. Springer, Heidelberg (2010)

[2] Andrews, G.R.: Concurrent programming: principles and practice. Benjamin-Cummings Publishing Co. Inc., Redwood City (1991)

[3] Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: a scalable memory allocator for multithreaded applications. SIGARCH Computer Architecture News 28(5), 117–128 (2000)

[4] Hanson, D.R.: C interfaces and implementations: techniques for creating reusable software. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)

[5] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, pp. 355–364. ACM, New York (2010)

[6] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Scalable flat-combining based synchronous queues. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 79–93. Springer, Heidelberg (2010)

[7] Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004, pp. 206–215. ACM, New York (2004)

[8] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13, 124–149 (1991)

[9] Scherer III, W.N., Lea, D., Scott, M.L.: A scalable elimination-based exchange channel. In: Workshop on Synchronization and Concurrency in Object-Oriented Languages, SCOOL 2005 (October 2005)

[10] Lea, D., Scherer III, W.N., Scott, M.L.: java.util.concurrent.exchanger source code (2011), `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/Exchanger.java`

[11] Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2005, pp. 253–262. ACM, New York (2005)

[12] Scherer III, W.N., Lea, D., Scott, M.L.: Scalable synchronous queues. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2006, pp. 147–156. ACM, New York (2006)

[13] Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks: preliminary version. In: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1995, pp. 54–63. ACM, New York (1995)

[14] Shavit, N., Zemach, A.: Diffracting trees. ACM Transactions on Computer Systems (TOCS) 14, 385–428 (1996)

[15] Shavit, N., Zemach, A.: Combining funnels: A dynamic approach to software combining. Journal of Parallel and Distributed Computing 60(11), 1355–1387 (2000)

[16] Treiber, R.K.: Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Research Center (2006)

Beeping a Maximal Independent Set

Yehuda Afek[1], Noga Alon[1,2], Ziv Bar-Joseph[3],
Alejandro Cornejo[4], Bernhard Haeupler[4], and Fabian Kuhn[5]

[1] The Blavatnik School of Computer Science, Tel Aviv University, 69978, Israel
[2] Sackler School of Mathematics, Tel Aviv University, 69978, Israel
[3] School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA 15213, USA
[4] CSAIL, Massachusetts Institute of Technology, MA 02139, USA
[5] Faculty of Informatics, University of Lugano, 6904 Lugano, Switzerland

Abstract. We consider the problem of computing a maximal independent set (MIS) in an extremely harsh broadcast model that relies only on carrier sensing. The model consists of an anonymous broadcast network in which nodes have no knowledge about the topology of the network or even an upper bound on its size. Furthermore, it is assumed that nodes wake up asynchronously. At each time slot a node can either beep (i.e., emit a signal) or be silent. At a particular time slot, beeping nodes receive no feedback, while silent nodes can only differentiate between none of its neighbors beeping, or at least one neighbor beeping.

We start by proving a lower bound that shows that in this model, it is not possible to locally converge to an MIS in sub-polynomial time. We then study four different relaxations of the model which allow us to circumvent the lower bound and compute an MIS in polylogarithmic time. First, we show that if a polynomial upper bound on the network size is known, it is possible to find an MIS in $\mathcal{O}(\log^3 n)$ time. Second, if sleeping nodes are awoken by neighboring beeps, then we can also find an MIS in $\mathcal{O}(\log^3 n)$ time. Third, if in addition to this wakeup assumption we allow beeping nodes to receive feedback to identify if at least one neighboring node is beeping concurrently (i.e., sender-side collision detection) we can find an MIS in $\mathcal{O}(\log^2 n)$ time. Finally, if instead we endow nodes with synchronous clocks, it is also possible to compute an MIS in $\mathcal{O}(\log^2 n)$ time.

1 Introduction

An MIS is a *maximal* set of nodes in a network such that no two of them are neighbors. Since the set is maximal every node in the network is either in the MIS or a neighbor of a node in the MIS. The problem of distributively selecting an MIS has been extensively studied in various models [3, 6, 21, 12, 14, 13, 15, 18, 16, 25] and has many applications in networking, and in particular in radio sensor networks. Some of the practical applications include the construction of a backbone for wireless networks, a foundation for routing and for clustering of nodes, and generating spanning trees to reduce communication costs [21, 25].

This paper studies the problem of computing an MIS in the discrete beeping wireless network model of [7]. The network is modeled as an undirected graph and time progresses in discrete and synchronous rounds, each being a time slot. In each round a node can either transmit a "jamming" signal (called a beep) or detect whether at least one neighbor beeps. We believe that such a model is minimalistic enough to be implementable in many real world scenarios. For example, it can easily be implemented using carrier sensing alone, where nodes only differentiate between silence and the presence of a signal on the wireless channel. Further, it has been shown that such a minimal communication model is strong enough to efficiently solve non-trivial tasks [2, 7, 19, 24]. The model is interesting from a practical point of view since carrier sensing typically uses less energy to communicate and reaches larger distances when compared with sending regular messages.

While this model is clearly useful for computer networks, it is also partially motivated by biological processes which are often more robust and adaptable than current computational systems. In biological systems, cells communicate by secreting certain proteins that are sensed ("heard") by neighboring cells [6]. This is similar to a node in a radio network transmitting a carrier signal which is sensed ("heard") by its neighbors. Such physical message passing allows for an upper bound on message delay. Thus, for a computational model based on these biological systems, we can assume a set of synchronous and anonymous processors communicating using beeps [7] in an arbitrary topology. We have recently shown that a variant of MIS is solved by a biological process, sensory organ precursor (SOP) selection in flies, and that the fly's solution provides a novel algorithm for solving MIS [2]. Here we extend algorithms for this model in several ways as discussed below.

This paper has two parts, we first prove a lower bound that shows that in a beeping model with adversarial wake-up it is not possible to locally converge on an MIS in sub-polynomial time. In the second part we present several relaxations of this model under which polylogarithmic MIS constructions are possible.

The lower bound shows that if nodes are not endowed with any information about the underlying communication graph, and their wake-up time is under the control of the adversary, any (randomized) distributed algorithm to find an MIS requires at least $\Omega(\sqrt{n/\log n})$ rounds. We remark that this lower bound holds much more generally. We prove the lower bound for the significantly more powerful radio network model with arbitrary message size and collision detection, and is therefore not an artifact of the amount of information which can be communicated in the beeping model.

Following the lower bound, in the second part of this paper four weaker adversarial models are considered and a polylog algorithm for MIS construction is presented for each. First, we present an algorithm that uses a polynomial upper bound on the size of the network, to compute an MIS in $\mathcal{O}(\log^3 n)$ rounds with high probability. Our next two algorithms assume that nodes are awakened by incoming messages (wake-on-beep). We present an $\mathcal{O}(\log^2 n)$ rounds algorithm in the beeping model with sender collision detection. Next, we present a $\mathcal{O}(\log^3 n)$ that works without sender collision detection in the same wakeup model. Finally, we show that even if nodes are only awakened by an adversary (and not by incoming messages) it is possible to use synchronous clocks to compute an MIS in $\mathcal{O}(\log^2 n)$ time without any information about the network. These results are summarized in the table below. We highlight that all the upper bounds presented in this paper compute an MIS eventually and almost surely, and thus only their running time is randomized.

	Assumptions	Running Time
Section 4	None (lower bound)	$\Omega(\sqrt{n/\log n})$
Section 5	Upper bound on n	$\mathcal{O}(\log^3 n)$
Section 6	Wake-on-Beep + Sender Collision Detection	$\mathcal{O}(\log^2 n)$
Section 7	Wake-on-Beep	$\mathcal{O}(\log^3 n)$
Section 8	Synchronous Clocks	$\mathcal{O}(\log^2 n)$

2 Related Work

The computation of an MIS has been recognized and studied as a fundamental distributed computing problem for a long time (e.g., [3, 4, 15, 20]). Perhaps the single most influential MIS algorithm is the elegant randomized algorithm of [3, 15], generally known as Luby's algorithm, which has a running time of $\mathcal{O}(\log n)$. This algorithm works in a standard message passing model, where nodes can concurrently reliably send and receive messages over all point-to-point links to their neighbors. [16] show how to improve the bit complexity of Luby's algorithm to use only $O(\log n)$ bits per channel ($O(1)$ bits per round). For the case where the size of the largest independent set in the 2-neighborhood of each node is restricted to be a constant (known as bounded independence or growth-bounded graphs), [23] presented an algorithm that computes an MIS in $\mathcal{O}(\log^* n)$ rounds. This class of graphs includes unit disk graphs and other geometric graphs that have been studied in the context of wireless networks.

While several methods were suggested for solving MIS in the case of symmetric processors, these methods have always assumed that nodes know something about the local or global topology of the network. Most previous methods assumed that nodes know the set of active neighbors each has at each stage of the execution. The first effort to design a distributed MIS algorithm for a wireless communication model in which the number of neighbors is not known is by [17]. They provided an algorithm for the radio network model with a $\mathcal{O}(\log^9 n/\log\log n)$ running time. This was later improved [18] to $\mathcal{O}(\log^2 n)$. Both algorithms assume that the underlying graph is a unit disk graph (the algorithms also work for somewhat more general classes of geometric graphs). In addition, while the algorithms solve the MIS problem in multi-hop networks with asynchronous wake up, they assume that an upper bound on the number of nodes in the network is known. In addition to the upper bound assumption their model allows for (and their algorithm uses) messages whose size is a function of the number of nodes in the network.

The use of carrier sensing (a.k.a. collision detection) in wireless networks has, e.g., been studied in [5, 10, 24]. As shown in [24], collision detection can be powerful and can be used to improve the complexity of algorithms for various basic problems. [22] show how to approximate a minimum dominating set in a physical interference (SINR) model where in addition to sending messages, nodes can perform carrier sensing. In [9], it is demonstrated how to use carrier sensing as an elegant and efficient way for coordination in practice.

The present paper is not he first one that uses carrier sensing alone for distributed wireless network algorithms. A similar model to the beep model considered here was first studied in [8, 19]. As used here, the model has been introduced in [7], where it is shown how to efficiently obtain a variant of graph coloring that can be used to schedule non-overlapping message transmissions. In [2] a variant of beeping model, there called *the fly model*, was considered. The fly model made three additional assumptions, which do not necessarily hold for biological systems: that all the processors wake up together at the same synchronous round, that a bound on the network size is known to the processors, and that processors can detect collisions. That is, processors can listen on the medium while broadcasting (as in some radio and local area networks). In addition to the work from [2] the most closely related work to this paper are results from [24]. In [24], it is shown that by solely using carrier sensing, an MIS can be computed in $O(\log n)$ time in growth-bounded graphs (a.k.a. bounded independence graphs). Here, we drop this restriction and study the MIS problem in the beeping model for general graphs.

3 Model

Following [7], we consider a synchronous communication network modeled by an arbitrary graph $G = (V, E)$ where the vertices V represent processors and the edges represent pairs of processors that can hear each other. We denote the set of neighbors of node u in G by $N_G(u) = \{v \mid \{u, v\} \in E\}$. For a node $u \in V$ we use $d_G(u) = |N_G(u)|$ to denote its degree (number of neighbors) and we use $d_{\max} = \max_{u \in V} d_G(u)$ to denote the maximum degree of G.

Instead of communicating by exchanging messages, we consider a more primitive communication model that relies entirely on carrier sensing. Specifically, in every round a participating process can choose to either beep or listen. If a process at node v listens in round t it can only distinguish between silence (i.e., no process $u \in N_{G_t}(v)$ beeps in round t) or the presence of one or more beeps (i.e., there exists a process $u \in N_{G_t}(v)$ that beeps in round t). Observe that a beep conveys less information than a conventional 1-bit message, since in that case it is possible to distinguish between no message, a message with a one, and a message with a zero.

Initially all processes are asleep, and a process starts participating the round after it is woken up by an adversary. We denote by $G_t \subseteq G$ the subgraph induced by the processes which are participating in round t.

Given an undirected graph H, a set of vertices $I \subseteq V(H)$ is an independent set of H if every edge $e \in E$ has at most one endpoint in I. An independent set

$I \subseteq V(H)$ is a maximal independent set of H, if for all $v \in V(H) \setminus I$ the set $I \cup \{v\}$ is not independent. An event is said to occur with high probability, if it occurs with probability at least $1 - n^{-c}$ for any constant $c \geq 1$, where $n = |V|$ is the number of nodes in the underlying communication graph. For a positive integer $k \in \mathbb{N}$ we use $[k]$ as short hand notation for $\{1, \ldots, k\}$. In a slight abuse of this notation we use $[0]$ to denote the empty set $\varnothing$ and for $a, b \in \mathbb{N}$ and $a < b$ we use $[a, b]$ to denote the set $\{a, \ldots, b\}$.

We say a (randomized) distributed algorithm solves the MIS problem in T rounds if any node irrevocably decides to be either inactive or in the MIS after being awake for at most T rounds. Furthermore, no two neighboring nodes decide to be in the MIS, and every node which decides to be inactive has at least one neighbor which decided to be in the MIS.

4 Lower Bound for Uniform Algorithms

In this section we show that without any additional power or a priori information about the network (e.g., an upper bound on its size or maximum degree), any fast-converging (randomized) distributed algorithm needs at least polynomial time to find an MIS with constant probability. In some ways, this result is the analog of the polynomial lower bound [11] on the number of rounds required for a successful transmission in the radio network model without collision detection or knowledge of n.

We stress that this lower bound is not an artifact of the beep model, but a limitation that stems from having message transmission with collisions and the fact that nodes are required to decide (but not necessarily terminate) without waiting until all nodes have woken up. Although we prove the lower bound for the problem of finding an MIS, this lower bound can be generalized to other problems (e.g., minimal dominating set, coloring, etc.).

Specifically, we prove the lower bound for the stronger communication model of local message broadcast with collision detection. In this communication model a process can choose in every round either to listen or to broadcast a message (no restrictions are made on the size of the message). When listening a process receives silence if no message is broadcast by its neighbors, it receives a collision if a message is broadcast by two or more neighbors, and it receives a message if it is broadcast by exactly one of its neighbors. The beep communication model can be easily simulated by this model (instead of beeping send a 1 bit message, and when listening translate a collision or the reception of a message to hearing a beep) and hence the lower bound applies to the beeping model.

At its core, our lower bound argument relies on the observation that a node can learn essentially no information about the graph G if upon waking up, it always hears collisions or silence. It thus has to decide whether it remains silent or beeps within a *constant* number of rounds. More formally:

Proposition 1. *Let A be an algorithm run by all nodes, and consider a fixed pattern $b \in \{\text{silent}, \text{collision}\}^*$. If after waking up a node u hears $b(r)$ whenever it listens in round r, then there are two constants $\ell \geq 1$ and $p \in (0, 1]$ that only* *depend on A and b such that either* **a)** *u remains listening indefinitely, or* **b)** *u listens for $\ell - 1$ rounds and broadcasts in round ℓ with probability p.*

Proof. Fix a node u and let $p(r)$ be the probability with which node u beeps in round r. Observe that $p(r)$ can only depend on r, what node u heard up to round r (i.e., b) and its random choices. Therefore, given any algorithm, either $p(r) = 0$ for all r (and node u remains silent forever), or $p(r) > 0$ for some r, in which case we let $p = p(r)$ and $\ell = r$.

We now prove the main result of this section:

Theorem 1. *If nodes have no a priori information about the graph G then any fast-converging distributed algorithm in the local message broadcast model with collision detection that solves the MIS problem with constant probability requires at least $\Omega(\sqrt{n/\log n})$ rounds.*

Proof. Fix any algorithm A. Using the previous proposition we split the analysis in three cases, and in all cases we show that with probability $1 - o(1)$ any algorithm runs for $\Omega(\sqrt{n/\log n})$ rounds.

We first ask what happens with nodes running algorithm A that hear only silence after waking up. Proposition 1 implies that either nodes remain silent forever, or there are constants ℓ and p such that nodes broadcast after ℓ rounds with probability p. In the first case, suppose nodes are in a clique, and observe that no node will ever broadcast anything. From this it follows that nodes cannot learn anything about the underlying graph (or even tell if they are alone). Thus, either no one joins the MIS, or all nodes join the MIS with constant probability, in which case their success probability is exponentially small in n.

Thus, for the rest of the argument we assume that nodes running A that hear only silence after waking up broadcast after ℓ rounds with probability p. Now we consider what happens with nodes running A that hear only collisions after waking up. Again, by Proposition 1 we know that either they remain silent forever, or there are constants m and p' such that nodes broadcast after m rounds with probability p'. In the rest of the proof we describe a different execution for each of these cases.

CASE 1: (A Node that Hears Only Collisions Remains Silent Forever) For some $k \gg \ell$ to be fixed later, we consider a set of $k-1$ cliques $C_1, \ldots, C_{k-1}$ and a set of k cliques $U_1, \ldots, U_k$, where each clique C_i has $\Theta(k \log n/p)$ vertices, and each clique U_j has $\Theta(\log n)$ vertices. We consider a partition of each clique C_i into k sub-cliques $C_i(1), \ldots, C_i(k)$ each with $\Theta(\log n/p)$ vertices. For simplicity, whenever we say two cliques are connected, they are connected by a complete bipartite graph.

Consider the execution where in round $i \in [k-1]$ clique C_i wakes up, and in round ℓ the cliques $U_1, \ldots, U_k$ wake up simultaneously. When clique U_j wakes up, it is is connected to sub-clique $C_i(j)$ for each $i < \ell$. Similarly, when clique C_i wakes up, if $i \geq \ell$ then for $j \in [k]$ sub-clique $C_i(j)$ is connected to clique U_j.

During the first $\ell - 1$ rounds only the nodes in C_1 are participating, and hence every node in C_1 broadcasts in round $\ell + 1$ with probability p. Thus, w.h.p., for all $j \in [k]$ at least two nodes in sub-clique $C_1(j)$ broadcast in round ℓ. This guarantees that all nodes in cliques $U_1, \ldots, U_k$ hear a collision during the first round they are awake, and hence they also listen for the second round. In turn, this implies that the nodes in C_2 hear silence during the first $\ell - 1$ rounds they participate, and again for $j \in [k]$, w.h.p., there are at least two nodes in $C_2(j)$ that broadcast in round $\ell + 2$.

By a straightforward inductive argument we can show (omitted) that in general, w.h.p., for each $i \in [k-1]$ and for every $j \in [k]$ at least two nodes in sub-clique $C_i(j)$ broadcast in round $\ell + i$. Therefore, also w.h.p., all nodes in cliques $U_1, \ldots, U_k$ hear collisions during the first $k - 1$ rounds after waking up.

Observe that at most one node in each C_i can join the MIS (i.e., at most one of the sub-cliques of C_i has a node in the MIS), which implies there exists at least one clique U_j that is connected to only non-MIS sub-cliques. However, since the nodes in U_j are connected in a clique, exactly one node of U_j must decide to join the MIS, but all the nodes in U_j have the same state during the first $k-1$ rounds. Therefore if nodes decide after participating for at most $k-1$ rounds, w.h.p., either no node in U_j joins the MIS, or more than two nodes join the MIS.

Finally since the number of nodes n is $\Theta(k^2 \log n + k \log n)$, we can let $k \in \Theta(\sqrt{n/\log n})$ and the claim follows.

CASE 2: (After Hearing Only Collisions, a Node Beeps with prob. p' After m Rounds)

For some $k \gg m$ to be fixed later let $q = \left\lfloor \frac{k}{4} \right\rfloor$ and consider a set of k cliques $U_1, \ldots, U_k$ and a set of $m-1$ cliques $S_1, \ldots, S_{m-1}$, where each clique U_i has $\Theta(\log n/p')$ vertices, and each clique S_i has $\Theta(\log n/p)$ vertices. As before, we say two cliques are connected if they form a complete bipartite graph.

Consider the execution where in round $i \in [m-1]$ clique S_i wakes up, and in round $\ell + j$ for $j \in [k]$ clique U_j wakes up. When clique U_j wakes up, if $j > 1$ it is connected to every U_i for $i \in \{\max(1, j-q), \ldots, j-1\}$ and if $j < m$ it is also connected to every clique S_h for $h \in \{j, \ldots, m\}$.

During the first $\ell - 1$ rounds only the nodes in S_1 are participating, and hence every node in S_1 broadcasts in round $\ell + 1$ with probability p, and thus, w.h.p., at least two nodes in S_1 broadcast in round $\ell + 1$. This guarantees the nodes in U_1 hear a collision upon waking up, and therefore they listen in round $\ell + 2$. In turn this implies the nodes in S_2 hear silence during the first $\ell - 1$ rounds they participate, and hence, w.h.p., at least two nodes in S_2 broadcast in round $\ell + 2$.

By a straightforward inductive argument we can show (omitted) that in general for $i \in [m-1]$ the nodes in S_i hear silence for the first $\ell - 1$ rounds they participate, and, w.h.p., at least two nodes in S_i broadcast in round $\ell + i$. Moreover, for $j \in [k]$ the nodes in U_j hear collisions for the first $m-1$ rounds they participate, and hence w.h.p. there are at least two nodes in U_j who broadcast in round $\ell + m + j - 1$. This implies that w.h.p. for $j \in [k-q]$ the nodes in U_j hear collisions for the first q rounds they participate.

We argue that if nodes choose weather or not to join the MIS q rounds after participating, then they fail w.h.p. In particular consider the nodes in clique U_j

for $j \in \{q, \ldots, k-2q\}$. These nodes will hear collisions for the first q rounds they participate, and they are connected to other nodes which also hear beeps for the first q rounds they participate. Therefore, if nodes decide after participating for less or equal than q rounds, w.h.p. either a node and all its neighbors won't be in the MIS, or two or more neighboring nodes join the MIS.

Finally since we have $n \in \Theta(m \log n + k \log n)$ nodes, we can let $k \in \Theta(n/\log n)$ and hence $q \in \Theta(n/\log n)$ and the theorem follows. □

5 Using an Upper Bound on n

To circumvent the lower bound, this section assumes that all nodes are initially given some upper bound $N > n$ (it is not required that all nodes are given the same upper bound) on the total number of nodes participating in the system. The algorithm described in this section guarantees that $\mathcal{O}(\log^2 N \log n)$ rounds after a node wakes up, it knows whether it belongs to the MIS or if it is inactive (i.e., covered by an MIS node). Therefore, if the known upper bound is polynomial in n (i.e., $N \in \mathcal{O}(n^c)$ for a constant c), then this algorithm solves the MIS problem in $\mathcal{O}(\log^3 n)$ rounds.

Algorithm: If a node hears a beep while listening at any point during the execution, it restarts the algorithm. When a node wakes up (or it restarts), it stays in an inactive state where it listens for $c \log^2 N$ consecutive rounds. After this inactivity period, nodes enter a competing state and group rounds into $\log N$ phases of $c \log N$ consecutive rounds. Due to the asynchronous wake up and the restarts, in general phases of different nodes will not be synchronized. In each round of phase i with probability $2^i/8N$ a node beeps, and otherwise it listens. Thus by phase $\log N$ a node beeps with probability $\frac{1}{8}$ in every round. After successfully going through the $\log N$ phases of activity (recall that when a beep is heard during any phase, the algorithm restarts) a node assumes it has joined the MIS and goes into a loop where it beeps in every round with probability 1/2 forever (or until it hears a beep).

Algorithm 1. MIS with an upper bound N on the size of the network

```
1: for $c \log^2 N$ rounds do listen                                   ▷ Inactive
2: for $i \in \{1, \ldots, \log N\}$ do                                ▷ Competing
3:     for $c \log N$ rounds do
4:         with probability $2^i/8N$ beep, otherwise listen
5: forever with probability $\frac{1}{2}$ beep then listen, otherwise listen then beep   ▷ MIS
```

Theorem 2. *Algorithm 1 solves the MIS problem in $O(\log^2 N \log n)$ time, where N is an upper bound on n that is a priori known to the nodes.*

This is another example which demonstrates that knowing a priori information about the network, even as simple as its size, can drastically change the complexity of a problem.

Proof Outline. First, we leverage the fact that for two neighboring nodes to go simultaneously into the MIS they have to choose the same actions (beep or listen) during at least $c \log N$ rounds. This does no happen w.h.p. and thus MIS nodes are independent w.h.p. On the other hand, since nodes which are in the MIS keep trying to break ties, an inactive node will never start competing while it has a neighbor in the MIS, and even in the low probability event that two neighboring nodes do join the MIS, one of them will eventually and almost surely leave the MIS. The more elaborate part of the proof is showing that w.h.p., any node either joins the MIS or has one of its neighbors in the MIS after $O(\log^2 N \log n)$ consecutive rounds. This requires three technical lemmas. First we show that if the sum of the beep probabilities of a neighbor are greater than a large enough constant, then they have been larger than a (smaller) constant for the $c \log N$ preceding rounds. We then use this to show that with constant probability, when a node u hears or produces beep, no neighbor of the beeping node beeps at the same time (and therefore the beeping node joins the MIS). Finally, since a node hears a beep or produces a beep every $O(\log^2 N)$ rounds, $\mathcal{O}(\log^2 N \log n)$ rounds suffice to stabilize w.h.p.

We remark that this algorithm is very robust, and in fact it works as-is if we give the adversary the power to crash an arbitrary set of nodes in every round. For such a powerful adversary, the running time guarantees have to be modified slightly, since if an inactive node has a single MIS node which is then crashed by the adversary, we must allow the inactive node to start competing to be in the MIS again.

Knowing When You Are Done. We've argued hat with high probability every node will (correctly) be in the MIS or the inactive state $\mathcal{O}(\log^3 n)$ rounds after waking up, however observe that the algorithm provides no way for a node to determine/output when it has arrived at the correct state. This is not a flaw of the algorithm, but an inherent limitation of the model and assumptions in which it is implemented. To see why, observe that regardless of what algorithm is used, in every round there is a non-zero probability that all nodes which are in the same state make the same random choices (beep or listen), and remain in the same state on the next round. Although this probability will drop exponentially fast with n, this already means that it is impossible for a node to distinguish with certainty between a solo execution and an execution in which it has one or more neighbors.

If we are willing to tolerate a small probability of error, we can turn Algorithm 1 (which is a Las Vegas algorithm) into a Monte Carlo algorithm and have every node output their state $\mathcal{O}(\log^3 n)$ rounds after waking up. Theorem 2 guarantees that w.h.p., the output describes an MIS. Another alternative, would be to endow nodes with unique identifiers encoded in $\mathcal{O}(\log n)$ bits. Using these identifiers it is possible to augment the last phase of the algorithm (i.e., line 5) to detect the case where two neighboring nodes are in the MIS state with certainty in asymptotically the same round complexity (we omit the details due to lack of space). Another alternative, which is described in detail in the next section, is to use sender-side collision detection.

6 Wake-on-Beep and Sender Collision Detection

This section considers a different relaxation of the model. Specifically, in addition to allowing the adversary to wakeup nodes arbitrarily, in this and the next section we assume that sleeping nodes wake up upon receiving a beep (wake-on-beep). Moreover this section we also assume that when a node beeps, it receives some feedback from which it can know if it beeped alone, or one of its neighbors beeped concurrently (a.k.a. sender collision detection). We will show that in such a model, it is possible to compute an MIS in $O(\log^2 n)$ time, even if there is no knowledge of the network (including no knowledge of neighbors and / or any upper bound on the network size).

This algorithm is an improvement of the algorithm presented in [2], which used an upper bound on the size of the network. In this algorithm nodes go through several iterations in which they gradually decrease the probability of being selected. The run time of the algorithm is still $O(\log^2 n)$ as we show below. Compared to the algorithm in [2], in addition to eliminating the dependence on any topological information, the current algorithm tolerates asynchronous wakeups if we assume wake-on-beep.

The algorithm proceeds in phases each consisting of x steps where x is the total number of phases performed so far (the phase counter). Assuming all nodes start at the same round, step i of each phase consists of two exchanges. In the first exchange nodes beep with probability $1/2^i$, and in the second exchange a node that beeped in the first exchange and did not hear a beep from any of its neighbors, beeps again, telling its neighbors it has joined the MIS and they should become inactive and exit the algorithm.

Asynchronous Wakeup. Nodes that wake up spontaneously, or are awakened by the adversary, propagate a wave of wake-up beeps throughout the network. Upon hearing the first beep, which must be the wake up beep, a node broadcasts the wake up beep on the next round, and then waits one round to ensure none of its neighbors is still waking up. This ensures that all neighbors of a node wake up either in the same round as that node or one round before or after that node. Due to these possible differences in wakeup time, we divide each exchange into 3 rounds. Nodes listen in all three rounds to incoming messages. During the second round of the first exchange each active node broadcasts a message to its neighbors with probability p_i (the value of p_i is given in the algorithm). The second exchange also takes three rounds. A node that broadcasts a message in the first exchange joins the MIS if none of its neighbors has broadcast in any of the three round of the first exchange. Such a node again broadcasts a message in the second round of the second exchange telling its neighbors to terminate the algorithm. The algorithm is detailed in Algorithm 2.

Safety Properties. While the algorithm in [2] uses a different set of coin flip probabilities, it relies on a similar two exchanges structure to guarantee the safety properties of the algorithm (when the algorithm terminates nodes are either MIS nodes or connected to a MIS node and no two MIS nodes are connected to each other). In [2], each exchange is only one round (since synchronous wakeup is

Algorithm 2. MIS with wake-on-beep and sender-side collision detection

```
 1: upon waking up (by adversary or beep) do beep to wake up neighbors
 2: wait for 1 round; $x \leftarrow 0$                          ▷ while neighbors wake up
 3: repeat
 4:     $x \leftarrow x + 1$                                     ▷ $2^x$ is current size estimate
 5:     for $i \in \{0, \ldots, x\}$ do                           ▷ $\log 2^x$ phases
 6:             ** exchange 1 ** with 3 rounds
 7:         listen for 1 round; $v \leftarrow 0$                   ▷ round 1
 8:         with probability $1/2^i$, beep and set $v \leftarrow 1$     ▷ round 2
 9:         listen for 1 round                                    ▷ round 3
10:         if received beep in any round of exchange 1 then $v \leftarrow 0$
11:             ** exchange 2 ** with 3 rounds
12:         listen for 1 round                                    ▷ round 1
13:         if $v = 1$ then beep and join MIS                      ▷ round 2
14:         listen for 1 round                                    ▷ round 3
15: until in MIS or received beep in any round of exchange 2
```

assumed). We thus need to show that replacing each one round exchange with a three round exchange does not affect the MIS safety properties of our algorithm. We will thus start by proving that the termination lemma from [2], which relies on the fact that all neighbors are using the same probability distribution in each exchange, still holds.

Lemma 1. *All messages received by node j in the first exchange of step i were sent by processors using the same probability as j in that step (see [1] for proof).*

Note that a similar argument would show that all messages received in the second exchange of step i are from processors that are in the second exchange of that step. Since our safety proof only relies on the coherence of the exchange they still hold for this algorithm. Notice also that by adding a listening round at the beginning and end of each exchange the algorithm now works in the un-slotted model (with at most doubling the round complexity).

Runtime Analysis. After establishing the safety guarantees, we next prove that with high probability all nodes terminate the algorithm in $O(\log^2 n)$ time where n is the number of nodes that participate in the algorithm. Let d_v be the number of *active* neighbors of node v. We start with the following definition [3]: a node v is Good if it has at least $d_v/3$ active neighbors u, s.t., $d_u \leq d_v$.

Lemma 4.4 from [3] : In every graph G at least half of the edges touch a Good vertex. Thus, $\sum_{v \in Good} d_v \geq |E|/2$.

Note that we need less than $O(\log^2 n)$ steps to reach $x \geq \log n$ since each phase until $x = \log n$ has less than $\log n$ steps. When $x \geq \log n$, the first $\log n$ steps in each phase are using the probabilities: $1, 1/2, 1/4, ..., 2/n, 1/n$. Below we show that from time $x = \log n$, we need at most $O(\log n)$ more phases to guarantee that all processors terminate with high probability.

Lemma 2. *The expected number of edges deleted in a phase (with more than* $\log n$ *steps) is* $\Omega(|E|)$

Proof. Fix a phase j, and fix a Good vertex v. We show that the expected number of edges incident to v that are deleted in this phase is $\Omega(d_v)$. Assume that at the beginning of phase j, $2^k \le d_v \le 2^{k+1}$ for some $k < \log n$. If when we reach step $i = k$ in phase j at least $d_v/20$ edges incident to v were already removed we are done. Otherwise, at step i there are still at least $d_v/3 - d_v/20 > d_v/4 \ge 2^{k-2}$ neighbors u of v with $d_u \le d_v$. Node v and all its neighbors u are flipping coins with probability $\frac{1}{2^k}$ at this step and thus the probability that at least one of them broadcasts is:

$$\Pr(v \text{ or a neighbor } u \text{ with } d_u \le d_v \text{ beeps}) \ge 1 - \left(1 - \frac{1}{2^k}\right)^{2^{k-2}} \cong 1 - 1/e^{1/4}.$$

On the other hand, all such nodes u, and v, have less than 2^{k+1} neighbors. Thus, the probability that a node from this set that broadcasts a message does not collide with any other node is:

$$\Pr(\text{no collisions}) \ge (1 - \frac{1}{2^k})^{2^{k+1}} \cong 1/e^2.$$

Thus, in every phase a Good node v has probability of at least $(1 - \frac{1}{e^{1/4}})\frac{1}{e^2} \ge \frac{1}{2^7}$ to be removed. Thus, the probability that v is removed is $\Omega(1)$ and hence the expected number of edges incident with v removed during this phase is $\Omega(d_v)$.

Since half the edges touch a Good node, by linearity of expectation the expected number of edges removed in each phase is $\ge \Omega(\sum_{v \in Good} d_v) = \Omega(|E|)$.

Note that since the number of edges removed in a phase in a graph (V, E) is clearly always at most $|E|$, the last lemma implies that for any given history, with probability at least $\Omega(1)$, the number of edges removed in a phase is at least a constant fraction of the number of edges that have not been deleted yet. Therefore there are two positive constants p and c, both bounded away from zero, so that the probability that in a phase at least a fraction c of the number of remaining edges are deleted is at least p. Call a phase successful if at least a fraction c of the remaining edges are deleted during the phase.

By the above reasoning, the probability of having at least z successful phases among m phases is at least the probability that a binomial random variable with parameters m and p is at least z. By the standard estimates for Binomial distributions, and by the obvious fact that $O(\log |E|/c) = O(\log n)$, starting from $x = \log n$ we need an additional $O(\log n)$ phases to finish the algorithm. Since each of these additional $O(\log n)$ phases consists of $O(\log n)$ steps, and since as discussed above until $x = \log n$ we have less than $O(\log^2 n)$ steps, the total running time of the algorithm is $O(\log^2 n)$. □

7 Wake-on-Beep with No Collision Detection

This section shows how to solve MIS in the wake-on-beep model with no collision detection. To extend our algorithm to a model with no collision detection we increase the number of exchanges in each step from 3 to x (x is the same as in Algorithm 2 and represents the current estimate of the network size). Prior to starting the exchanges in each step each *active* processor flips a coin with the same probability as in Algorithm 2. If the flip outcome is 0 (tail) the processor only listens in the next cx exchanges (for a constant c discussed below). If the flip outcome is 1 the processor sets $v = 1$ and sets, with probability 0.5, every entry in the vector X of size cx to 1 (the rest are 0). In the following exchanges the processor broadcasts a beep if $X(j) = 1$ where j is the index of that exchange and only listens if $X(j) = 0$. If at any of the exchanges it listens it hears a beep it sets $v = 0$ and stops broadcasting (even in the selected exchanges). If a node hears a beep during these exchanges it does not exit the algorithm. Instead, it denotes the fact that one of its neighbors beeped and sets itself to be inactive. If it does not hear a beep in any of the exchanges of a future phase it becomes active and continues as described above. Similarly, a node that beeped and did not hear any beep in a specific step (indicating that it can join the MIS) continues to beep indefinitely (by selecting half the exchanges in all future steps and following the algorithm above).

However, the guarantees we provide differ from those in the stronger collision detection model. Specifically, the algorithm guarantees that after a certain time (which depends on the network size and is derived below), all MIS members are fixed, and the safety requirements hold (all nodes not in the MIS are connected to a node in the MIS and no two MIS members are connected). Until this time, nodes can decide to become MIS members and later drop from the set if they find out that one of their neighbors has also decided to join the MIS. Since nodes do not have an estimate of the network size the processors continue to perform the algorithm indefinitely. At the end of the section we describe a stopping criteria that could be used if an estimate of the network size were available.

The main difference between this algorithm and Algorithm 2 a set of competition exchanges that are added at the end of each coin flip. The number of competition exchanges is equal to the current phase counter (which serves as the current estimate of the network size). Initially the competition rounds are short and so they would not necessarily remove all collisions. We require that nodes that attempt to join continue to participate in all future competition rounds (when $v = 1$). Processors that detect a MIS member as a neighbor set z to 1 and do not broadcast until they go through one complete set of competition exchanges in which they do not hear any beep. If and when this happens they set $z = 0$ and become potential MIS candidates again.

While not all collisions will be resolved at the early phases, when $x \geq \log n$ each set of competition exchanges is very likely to remove all collisions. We prove below that once we arrive at such x values, all collisions are resolved with very high probability such that only one processor in a set of colliding processors remains with $v = 1$ at the end of these competition exchanges. From there, it takes another $O(\log n)$ phases to select all members of the MIS as we have shown for Algorithm 1. Since each such phase takes $O(\log n)$ steps with each step

Algorithm 3. MIS with wake-on-beep *without* sender-side collision detection

```
1: upon waking up (by adversary or beep) do beep to wake up neighbors
2: wait for 1 round; $x \leftarrow 0$; $v \leftarrow 0$; $z \leftarrow 0$      ▷ while neighbors wake up
3: repeat forever
4:     $x \leftarrow x + 1$
5:     for $i \in \{0, \ldots, x\}$ do
6:         if $v = 0 \wedge z = 0$ then with probability $1/2^i$ set $v \leftarrow 1$
7:         $X \leftarrow$ random 0/1-vector of length $cx$      ▷ $c$ is a constant
8:         $z \leftarrow 0$
9:           ** $cx$ competition exchanges **
10:        for $k \in \{1, \ldots, cx\}$ do
11:            listen for 1 round
12:            if beep received then $v \leftarrow 0$; $z \leftarrow 1$      ▷ $z = 1$: conn. to node in MIS
13:            if $v = 0 \vee X[k] = 0$ then
14:                listen for 1 round; if beep received then $v \leftarrow 0$; $z \leftarrow 1$
15:            else      ▷ $v = 1 \wedge X[k] = 1$
16:                beep for 1 round
17:            listen for 1 round; if beep received then $v \leftarrow 0$;
```

taking $O(\log n)$ rounds for the competition, the total run time of the algorithm is $O(\log^3 n)$.

Below we prove that if two neighbors in the network have $v = 1$ after the coin flip in step i in a phase with $x \geq \log n$, then w.h.p., one would set $v = 0$ at the end of step i of that phase and so at most one of them enters the MIS.

Lemma 3. *Assume processor y collided with one or more of its neighbors setting $v = 1$ in step i of phase $x \geq \log n$. Then the probability that y would still be colliding with any of its neighbors at the end of the cx competition exchanges for step i is $\leq \frac{1}{n^{c/3}}$.*

Proof. If at any of the exchanges in this step all neighbors of y have $v = 0$ we are done. Otherwise in each exchange, with probability at least $1/4$ y decided not to broadcast whereas one of its colliding neighbors decided to broadcast. Thus, the probability that y does not resolve its collision in a specific exchange is $\leq 3/4$. Since there are $(c \log n)$ exchanges in this step, the probability that y is still colliding at the end of these competition exchanges $\leq (\frac{3}{4})^{c \log n} \leq \frac{1}{n^c/3}$. □

Note that if a collision is not resolved in a specific exchange the colliding nodes continue to broadcast in the following phase. As we proved in the previous section, if all collisions are resolved in the $O(\log n)$ phases that follow the phase $x = \log n$ the algorithm will result in a MIS set with very high probability. Since we only need $O(\log^2 n) < n$ steps for this, and we have n nodes, the probability that there exists a step and a node in phase $x \geq \log n$ such that a node that collided during this step with a neighbor does not resolve this collision in that step is smaller than $\frac{1}{n^{c/3-2}}$. Thus, with probability $\geq 1 - \frac{1}{n^{c/3-2}}$ all collisions are detected and the MIS safety condition holds.

Stopping Criteria When an Upper Bound on the Network Size Exists. The above algorithm leads to a MIS set and does not require knowledge of the network size. However, the time it takes to reach an MIS is a function of the size of the network and so if nodes do not have an estimate of this number they cannot terminate the algorithm and need to indefinitely listen to incoming messages. Note that, as the analysis above indicates, if a rough estimate on the network size n exists, nodes can terminate the algorithm when $x = 2\log n + 1$. At that phase we have a high probability that every collision that has occurred during the last $\log n$ phases has been resolved ($\geq 1 - \frac{1}{n^{c/3-2}}$) and as proved in the previous section when all collisions are resolved the algorithm terminates with very high probability. Note that when using the upper bound the algorithm is no longer Las Vegas, but rather Monte Carlo since there is a (very low) probability that the algorithm terminates with two neighbors that are both in the MIS.

8 Synchronized Clocks

For this section the only assumption we make on top of the beep model is that that nodes have synchronized clocks, i.e., know the current round number t.

Algorithm: Nodes have three different internal states: *inactive*, *competing*, and *MIS*. Each node has a parameter k that is monotone increasing during the execution of the algorithm. All nodes start in the inactive state with $k = 6$.

Nodes communicate in beep-triples, and synchronize by starting a triple only when $t \equiv 0 \pmod 3$. The first bit of the triple is the Restart-Bit. A beep is sent for the Restart-Bit if and only if $t \equiv 0 \pmod k$. If a node hears a beep on its Restart-Bit it doubles its k and if it is active it becomes inactive. The second bit sent in the triple is the MIS-Bit. A beep is sent for the MIS-Bit if and only if a node is in the MIS state. If a node hears a beep on the MIS-bit it becomes inactive. The last bit send in a triple is the Competing-Bit. If inactive, a node listens to this bit, otherwise it sends a beep with probability 1/2. If a node hears a beep on the Competing-Bit it becomes inactive. Furthermore, if a node is in the MIS-state and hears a beep on the Competing-Bit it doubles its k. Lastly, a node transitions from inactive to active between any time t and $t+1$ for $t \equiv 0 \pmod k$. Similarly, if a node is active when $t = 0 \mod k$ then it transitions to the MIS state. In the sequel, we refer to this algorithm as Algorithm 2. The state transitions are also depicted in Figure 1.

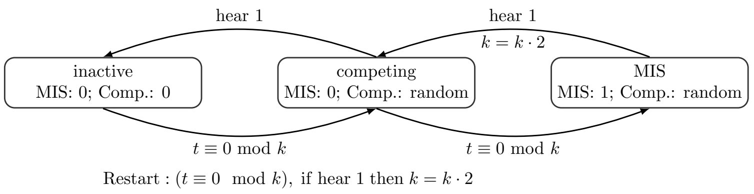


Fig. 1. State Diagram for Algorithm 2

Idea: The idea of the algorithm is to employ Luby's permutation algorithm in which a node picks a random $O(\log n)$-size priority which it shares with its neighbors. A node then joins the MIS if it has the highest priority among its neighbors, and all neighbors of an MIS node become inactive. Despite the fact that this algorithm is described for the message exchange model, it is straightforward to adapt the priority comparisons to the beep model. For this, a node sends its priority bit by bit, starting with the highest-order bit and using a beep for a 1. The only further modification is that a node stops sending its priority if it has already heard a beep on a higher order bit during which it remained silent because it had a zero in the corresponding bit. Using this simple procedure, a node can easily realize when a neighboring node has a higher priority. Furthermore, a node can observe that it has the highest-priority in its neighborhood which is exactly the case if it does not hear any beep .

Therefore, as long as nodes have a synchronous start and know n (or an upper bound) it is straightforward to get Luby's algorithm working in the beep model in $O(\log^2 n)$ rounds.

In the rest of this section we show how to remove the need for an upper bound on n and a synchronous start. We solely rely on synchronized clocks to synchronize among nodes when a round to transmits a new priority starts. Our algorithm uses k to compute an estimate for the required priority-size $O(\log n)$. Whenever a collision occurs and two nodes tie for the highest priority the algorithm concludes that k is not large enough yet and doubles its guess. The algorithm uses the Restart-Bit to ensure that nodes locally work with the same k and run in a synchronized manner in which priority comparisons start at the same time (namely every $t \equiv 0 \pmod{k}$). It is not obvious that either a similar k or a synchronized priority comparison is necessary but it turns out that algorithms without them can stall for a long time. In the first case this is because repeatedly nodes with a too small k enter the MIS state simultaneously while in the second case many asynchronously competing nodes (even with the same, large enough k) keep eliminating each other without one becoming dominant and transitioning into the MIS state.

Analysis: To proof the algorithm's correctness, we first show two lemmas that show that with high probability k cannot be super-logarithmic.

Lemma 4. *With high probability $k \in O(\log n)$ for all nodes during the execution of the algorithm.*

Proof. We start by showing that two neighboring nodes u, v in the MIS state must have the same k and transitioned to the MIS state at the same time. We prove both statements by contradiction.

For the first part assume that nodes u and v are in the MIS state but u transitioned to this state (the last time) before v. In this case v would have received the MIS-bit from u and become inactive instead of joining the MIS, a contradiction.

Similarly, for sake of contradiction, we assume that $k_u < k_v$. In this case, during the active phase of u before it transitioned to the MIS at time t it would have sent a beep at time $t - k_u$ and thus would have become inactive, contradicting the assumption that $k_u < k_v$.

Given this we now show that for a specific node u it is unlikely to become the first node with a too large k. For this we note that k_u gets doubled because of a Restart-Bit only if a beep from a node with a larger k is received. This node can therefore not be responsible for u becoming the first node getting a too large k. The second way k can increase is if a node transitions out of the MIS state because it receives a Competing-Bit from a neighbor v. In this case, we know that u competed against at least one such neighbor for $k/6$ phases with none of them loosing. The probability of this to happen is $2^{-k/6}$. Hence, if $k \in \Theta(\log n)$, this does not happen w.h.p. A union bound over all nodes and the polynomial number of rounds in which nodes are not yet stable finishes the proof. □

Theorem 3. *If during an execution the $O(\log n)$ neighborhood of a node u has not changed for $\Omega(\log^2 n)$ rounds then u is stable, i.e., u is either in the MIS state with all its neighbors being inactive or it has at least one neighbor in the MIS state whose neighbors are all inactive.*

Proof. First observe that if the whole graph has the same value of k and no two neighboring nodes transition to the MIS state at the same time, then our algorithm behaves exactly as Luby's original permutation algorithm, and therefore terminates after $O(k \log n)$ rounds with high probability. From a standard locality argument, it follows that a node u also becomes stable if the above assumptions only hold for a $O(k \log n)$ neighborhood around u. Moreover, since Luby's algorithm performs only $O(\log n)$ rounds in the message passing model, we can improve our locality argument to show that in if a $O(\log n)$ neighborhood around u is well-behaved, then u behaves as in Luby's algorithm.

Since the values for k are monotone increasing and propagate between two neighboring nodes u, v with different k (i.e., $k_u > k_v$) in at most $2k_u$ steps, it follows that for a node u it takes at most $O(k_u \log n)$ rounds until either k_u increases or all nodes v in the $O(\log n)$ neighborhood of u have $k_v = k_u$ for at least $O(k \log n)$ rounds. We can furthermore assume that these $O(k \log n)$ rounds are collision free (i.e, no two neighboring nodes go into the MIS), since any collision leads with high probability within $O(\log n)$ rounds to an increased k value for one of the nodes.

For any value of k, within $O(k \log n)$ rounds a node thus either performs Luby's algorithm for $O(\log n)$ priority exchanges, or it increases its k. Since k increases in powers of two and, according to Lemma 4, with high probability it does not exceed $O(\log n)$, after at most $\sum_i^{O(\log \log n)} 2^i \cdot 3 \cdot O(\log n) = O(\log^2 n)$ rounds the status labeling around a $O(\log n)$ neighborhood of u is a proper MIS. This means that u is stable at some point and it is not hard to verify that the function of the MIS-bit guarantees that this property is preserved for the rest of the execution. □

We remark that as the algorithm of Section 5, this algorithm is also robust enough to work as-is with an adversary capable of crashing nodes (with the same caveats on the guarantees mentioned in Section 5).

References

[1] Afek, Y., Alon, N., Bar-Joseph, Z.: Beeping on the fly (arxiv version) (2011), http://arxiv.org/abs/1106.2126v1

[2] Afek, Y., Alon, N., Barad, O., Hornstein, E., Barkai, N., Bar-Joseph, Z.: A biological solution to a fundamental distributed computing problem. Science 331(6014), 183–185 (2011)

[3] Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. Journal of Algorithms (1986)

[4] Awerbuch, B., Goldberg, A.V., Luby, M., Plotkin, S.A.: Network decomposition and locality in distributed computation. In: Proc. of the 30th Symposium on Foundations of Computer Science (FOCS), pp. 364–369 (1989)

[5] Chlebus, B., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in unknown radio networks. In: Prof. 11th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 861–870 (2000)

[6] Collier, J.R., Monk, N.A., Maini, P.K., Lewis, J.H.: Pattern formation by lateral inhibition with feedback: a mathematical model of delta-notch intercellular signalling. J. Theor. Biol. 183(4), 429–446 (1996)

[7] Cornejo, A., Kuhn, F.: Deploying wireless networks with beeps. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 148–162. Springer, Heidelberg (2010)

[8] Degesys, J., Rose, I., Patel, A., Nagpal, R.: Desync: self-organizing desynchronization and TDMA on wireless sensor networks. In: Prof. 6th Conf. on Information Processing in Sensor Networks (IPSN), p. 20 (2007)

[9] Flury, R., Wattenhofer, R.: Slotted programming for sensor networks. In: Proc. 9th Conference on Information Processing in Sensor Networks, IPSN (2010)

[10] Ilcinkas, D., Kowalski, D., Pelc, A.: Fast radio broadcasting with advice. Theoretical Computer Science 411(14-15) (2010)

[11] 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)

[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] Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, pp. 300–309 (2004)

[14] Kuhn, F., Moscibroda, T., Wattenhofer, R.: The price of being near-sighted. In: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, pp. 980–989 (2006)

[15] Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing 15, 1036–1053 (1986)

[16] Métivier, Y., Michael Robson, J., Saheb-Djahromi, N., Zemmari, A.: An optimal bit complexity randomized distributed MIS algorithm. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 323–337. Springer, Heidelberg (2010)

[17] Moscibroda, T., Wattenhofer, R.: Efficient computation of maximal independent sets in structured multi-hop radio networks. In: Proc. of 1st International Conference on Mobile Ad Hoc Sensor Systems, MASS (2004)

[18] Moscibroda, T., Wattenhofer, R.: Maximal Independent Sets in Radio Networks. In: Proc. 24th Symposium on Principles of Distributed Computing, PODC (2005)

[19] Motskin, A., Roughgarden, T., Skraba, P., Guibas, L.: Lightweight coloring and desynchronization for networks. In: Proc. 28th IEEE Conf. on Computer Communications, INFOCOM (2009)

[20] Panconesi, A., Srinivasan, A.: On the complexity of distributed network decomposition. Journal of Algorithms 20(2), 581–592 (1995)

[21] Peleg, D.: Distributed computing: a locality-sensitive approach. Society for Industrial and Applied Mathematics, Philadelphia (2000)

[22] 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 Symposium on Mobile Ad Hoc Networking and Computing, MOBIHOC (2008)

[23] Schneider, J., Wattenhofer, R.: A Log-Star Maximal Independent Set Algorithm for Growth-Bounded Graphs. In: Proc. 28th Symposium on Principles of Distributed Computing, PODC (2008)

[24] Schneider, J., Wattenhofer, R.: What is the use of collision detection (in wireless networks)? In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 133–147. Springer, Heidelberg (2010)

[25] Wan, P.-J., Alzoubi, K.M., Frieder, O.: Distributed construction of connected dominating set in wireless ad hoc networks. Mobile Networks and Applications (2004)

Trading Bit, Message, and Time Complexity of Distributed Algorithms

Johannes Schneider and Roger Wattenhofer

Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland

Abstract. We present tradeoffs between time complexity t, bit complexity b, and message complexity m. Two communication parties can exchange $\Theta(m \log(tb/m^2) + b)$ bits of information for $m < \sqrt{bt}$ and $\Theta(b)$ for $m \geq \sqrt{bt}$. This allows to derive lower bounds on the time complexity for distributed algorithms as we demonstrate for the MIS and the coloring problems. We reduce the bit-complexity of the state-of-the art $O(\Delta)$ coloring algorithm without changing its time and message complexity. We also give techniques for several problems that require a time increase of t^c (for an arbitrary constant c) to cut both bit and message complexity by $\Omega(\log t)$. This improves on the traditional time-coding technique which does not allow to cut message complexity.

1 Introduction

The efficiency of a distributed algorithm is assessed with at least one out of three classic distributed complexity measures: time complexity (number of rounds for synchronous algorithms), communication or bit complexity (total number of bits transmitted), and message complexity (total number of messages transmitted). Depending on the application, one or another measure might be more relevant. Generally speaking, time complexity has received most attention; but communication complexity (bandwidth constraints) or message complexity (accounting for message overhead) play a vital role as well. One cannot just ignore one of the measures, as there are tradeoffs: One may for instance sometimes cut down on time by exchanging larger messages. Alternatively, one may save messages and bits by communicating "silently". Two parties may for instance communicate for free by telephone by simply never picking up the phone, and instead letting the phone ring for a long time when transmitting a binary 1, and just a short time for a binary 0. A more sophisticated example for silent communication employs time-coding to communicate information through time. As illustration consider pulse-position modulation, as used in wireless and optical communication. A k-bit message can be dispersed over time by encoding the message with a single pulse in one of 2^k possible slots. Employing a single pulse within time t allows to communicate at most $\log t$ bits.[1] Reducing message complexity is harder in general, and in some cases impossible as there are dependencies between messages.

[1] The amount of information that can be communicated follows directly from our bound.

In this paper, we identify mechanisms for symmetry breaking that cut both message and bit complexity by a factor of $\log t$, even though multiple messages cannot be combined into a single message through time-coding.

Although it is well-known that these dependencies exist, to the best of our knowledge the tradeoffs are not completely understood. A considerable amount of work deals with both message size and time complexity. These two measures give a (rough) bound on the bit complexity, e.g. time multiplied by message size as an upper bound. However, we show that for a given fixed bit complexity allowing many arbitrary small messages (i.e. consisting of 1 bit) compared to allowing only one large message might cause a drastic (up to exponential) gain in time. For some examples both the time and overall message complexity, i.e. the messages transmitted by all nodes, have been optimized, but the tradeoffs between all three have not been investigated to the best of our knowledge.

In the first part of the paper we answer questions like "If we can prolong an algorithm by a factor t in time and can increase the number of messages by a factor m, what is the effect on the bit complexity b?" We give a tight bound on the amount of information exchangeable between two nodes of $\Theta(m \log(tb/m^2) + b)$ bits for $m < \sqrt{bt}$ and $\Theta(b)$ for larger m. A bound on the (communicable) information together with a bound on the minimum required information that has to be exchanged to solve a problem yields bounds on the time-complexity. We derive such bounds for typical symmetry breaking problems, such as coloring and maximal independent sets. We show that for $t \in [2, n]$ any MIS and $O(\Delta)$ coloring algorithms using up to $c_0 \log n / \log t$ bits and messages for a constant c_0 require time t. In light of the state of the art upper bounds for unrestricted communication of $\mathrm{O}(\log n)$ using $O(\log n)$ bits for the MIS problem, and $O(\log^* n)$ for the $O(\Delta)$ coloring problem, our lower bound indicates that even a logarithmic factor of $\log t$ in the amount of transmittable bits can make more than an exponential difference in time.

In the second part, we identify two coding schemes, i.e. transformations, to gain a factor $\log t$ in bit as well as message complexity by a time increase of t^c that cannot be achieved with traditional time coding. We employ them to deterministic and randomized coloring and maximal independent set algorithms. Our techniques are applicable beyond these problems, e.g. for certain divide-and-conquer algorithms. We also improve the bit complexity for the fastest randomized $\mathrm{O}(\Delta + \log^{1+1/\log^* n} n)$ coloring for Δ being at least polylogarithmic, i.e. from $\mathrm{O}(\log \Delta \log n)$ to $\mathrm{O}(\log n \log \log n)$, while maintaining its time complexity.

2 Related Work

In [7] the notion of "bit rounds" was introduced in the context of a coloring algorithm, where a node must transmit either 0 or 1 in one bit round. This *bit round complexity* is an interesting hybrid between time and bit complexity, particularly useful in systems where sending a single bit does not incorporate a significant protocol overhead. The paper [7] also states a lower bound of $\Omega(\log n)$ bit rounds to compute a coloring on a ring. In contrast, our bit-time complexity model is motivated by the fact that in practice time is frequently divided into slots, i.e. rounds, and nodes might not transmit at all in a slot or they might transmit more than a single bit. In some sense, the bit-time complexity model unifies algorithm complexity focusing on running time and communication complexity. For a survey on communication complexity see [11]. In a common set-up two (or more) parties A, B want to compute a function f that depends on values held by A and B and the goal is to derive bounds on the needed information. For example, [16] shows that for some network topologies (involving more than two nodes) reducing the number of allowed message exchanges by 1 can exponentially increase the time complexity. The amount of exchangeable information between two parties given a certain number of messages and time can be found in e.g. [17]. To the best of our knowledge, we are the first to extend the tradeoff to allow for a variable number of bits per message.

In [3] the communication complexity of breaking symmetry in rings and chains is investigated. In [4] the running time of MIS algorithms is investigated depending on the amount of advice (measured in bits) that nodes are given before the start of the algorithm. For a ring graph it is shown that $\Omega(n/\log^{(k)} n)$ bits of information are needed for any constant k to break the lower bound of $\Omega(\log^* n)$ [12]. At first, asking a global instance, knowing the exact topology of the graph, for advice seems to contradict the distributed approach. But the question regarding the amount of needed information to solve a task is interesting and valuable for enhancing the understanding of distributed algorithms. In particular, since some (aggregate) knowledge of the graph is often necessary for an efficient computation, e.g. the type of the graph.

There is a myriad of papers for different problems that consider two complexity measures. However, whereas many papers concentrate on time complexity and merely mention message size [14,13,8,1,2], others derive explicit tradeoffs [5,15,6].

In a paper by Métivier et al. [15] an algorithm for the MIS problem was stated running in time $\mathrm{O}(\log n)$ with bit complexity $\mathrm{O}(\log n)$ for general graphs. It improves on the bit complexity of the fastest algorithm [14]. Essentially, each node draws a random number in $[0, n]$ and is joined the MIS, if its number is the smallest. Our MIS algorithm trading time for bit/message complexity improves on [14] through a different technique. For the MIS problem arbitrary large messages do not allow for an arbitrary fast algorithm, i.e. in general graphs every algorithm requires at least $\Omega(\sqrt{\log n/\log\log n})$ or $\Omega(\log \Delta/\log\log \Delta)$ communication rounds for computing a MIS [9]. Interestingly, the opposite is true: An arbitrarily slow algorithm allows for constant message and bit complexity. The lower bound is achieved with a pseudo-symmetric graph, such that a node needs to get to know its neighborhood up to distance $\Omega(\sqrt{\log n/\log\log n})$ or $\Omega(\log \Delta/\log\log \Delta)$. For the coloring problem, [18] presented a technique, where nodes select multiple colors and keep any of them, if it is not also selected by a neighbor. The bit complexity for $\Delta + 1$ as well as $O(\Delta)$ algorithms is $\mathrm{O}(\log \Delta \log n)$. The latter runs in time $\mathrm{O}(\log^* n)$ for graphs with $\Delta \in \Omega(\log^{1+1/\log^* n} n)$. Recently, [19] used [18] to derive a $(1 - 1/O(\chi))(\Delta + 1)$ coloring, where χ denotes the chromatic number of the graph. Deterministic $\Delta + 1$ coloring algorithms [1,8] are faster for graphs of sublogarithmic degrees, i.e. need $O(\Delta + \log^* n)$ time and might require a message exchange of size $O(\log n)$ in a communication round.

Apart from applications that rely only on pure time-coding, in [20] a data gathering protocol is implemented saving on energy and bandwidth by coding messages partly through time.

3 Model and Definitions

The communication network is modeled with a graph $G = (V, E)$. For each node, there exists a distinct communication channel (edge) to each neighbor. Initially, a node v only knows the number of neighbors $|N(v)|$ and has no other information about them. We use the message passing model, i.e. each node can exchange one distinct message with each of its neighbors in one synchronous communication round. Communication is error free. All nodes start executing the algorithm concurrently. The time complexity denotes the number of rounds until the last node terminates.

In a (vertex) coloring any two neighboring nodes u, v have a different color. A set $T \subseteq V$ is said to be independent in G if no two nodes $u, v \in T$ are neighbors. A set $S \subseteq V$ is a maximal independent set (MIS), if S is independent and there exists no independent superset $T \supset S$. In our algorithm a node remains *active* as long as it might still join the MIS, i.e. as long as it has no neighbor in the MIS and it is not in the MIS itself. *Inactive* nodes are removed from the graph G. For a node v its neighborhood $N^r(v)$ represents all active nodes within r hops of v (not including v itself). We use $N(v)$ for $N^1(v)$. The r hop neighborhood $N^r(v)$ including v is denoted by $N^r_+(v)$. The term "with high probability" abbreviated by w.h.p. denotes the number $1-1/n^c$ for an arbitrary constant c. The maximum degree is denoted by Δ and $\Delta_{N_+(v)}$ denotes the maximum degree of a node in $N_+(v)$, i.e. $\Delta_{N_+(v)} := \max_{u \in N_+(v)} d(u)$.

The bit complexity denotes the maximum sum of the number of bits transmitted over any edge during the execution of the algorithm, i.e. if an algorithm has time complexity t then the bit complexity is $\max_{e \in E} \sum_{r=0}^{t-1} b_e(r)$, where $b_e(r)$ denotes the number of bits transmitted over edge e in round r. Analogously, the message complexity denotes the maximum number of messages transmitted over any edge.

The time complexity of a distributed algorithm is traditionally defined as the number of communication rounds until the *last* node completes the algorithm. Somewhat inconsistently, message respectively bit complexity often measure the *total* of all exchanged messages respectively bits (of all nodes) or the expectation of message and bits exchanges of a single node during the execution. In this paper, analogous to the definition of time complexity, we consider the worst node only; in other words, the message respectively bit complexity is given by the number of messages or bits exchanged by the most loaded node. Both views have their validity, are commonly used and sometimes even coincide. The focus on a more local "maximum" measure is motivated by the observation that for distributed systems, an individual node might often form a bottleneck, and delay an algorithm, although overall the constraints on bandwidth, energy etc. are fulfilled. For example, if a single node in a battery powered sensor network must transmit much more often than other nodes, it will become non-operational much more quickly. This might have devastating effects on the network topology, e.g. disconnecting it and thereby preventing further data aggregation.

4 Tight Bounds on the Transmittable Information

In all (reasonable) distributed algorithms nodes must exchange a certain minimum of information with their neighbors. The amount of exchanged information is not only given by the total amount of bits contained in the messages, but also by the times, when the messages were sent and the size of the messages. In other words, the number of different (observable) behaviors of a node v by a neighbor u, i.e. the number of different ways v can communicate with u, determines the total amount of exchanged information between two nodes. We first bound the number of exchangeable information between two communication parties. By using a lower bound for the minimum needed amount of exchanged information for any kind of problem one therefore gets a lower bound on the time complexity of any algorithm depending on the bits and messages exchanged. We illustrate this general technique by deriving lower bounds for the MIS and coloring problem.

Theorem 1. *If a node executes t rounds using up to $m \leq t$ messages (at most one message per round) with a total of $b \geq m$ bits within messages (i.e. at least one bit per message), it can communicate in total $\Theta(m \log(tb/m^2) + b)$ bits for $m < \sqrt{bt}$ and $\Theta(b)$ bits for $m \geq \sqrt{bt}$.*

Proof. A node can decide not to transmit at all or it can transmit in $n_r \in [1, m]$ rounds $\{r_1, r_2, ..., r_{n_r}\}$ with $r_i \in [0, t-1]$ for $1 \leq i \leq n_r$. In each chosen round r_i the node transmits at least one bit. The total number of choices of rounds is given by $\binom{t}{n_r}$. Say a node wants to transmit n_{r_i} bits in round r_i then the sum of all bits transmitted n_t in all rounds must be at least n_r and at most b, i.e. $n_r \leq n_t := \sum_{i=‘}^{n_r} n_{r_i} \leq b$. Thus the number of all possible sequences $(n_{r_1}, n_{r_2}, ..., n_{r_{n_r}})$ with $n_{r_i} \in [1, b - n_r + 1]$ is given by the composition of n_t into exactly n_r parts, i.e. the number of ways we can write n_t as a sum of n_r terms, i.e. $\binom{n_t - 1}{n_r - 1} \leq \binom{n_t}{n_r}$. Each of the at most n_t transmitted bits can either be 0 or 1, yielding 2^{n_t} combinations. Multiplying, these three terms and adding one for the case that a node does not transmit, i.e. $\binom{t}{n_r} \cdot \binom{n_t-1}{n_r-1} \cdot 2^{n_t} + 1$, gives a bound on the different behaviors of a node for a fixed n_r. Thus, overall the number of behaviors is upper bounded by:

$$1 + \sum_{n_r=1}^{m} \sum_{n_t=n_r}^{b} \binom{t}{n_r} \cdot \binom{n_t}{n_r} \cdot 2^{n_t} \leq 1 + mb \cdot \max_{1 \leq n_r \leq m, 0 \leq n_t \leq b} \binom{t}{n_r} \cdot \binom{n_t}{n_r} \cdot 2^{n_t}$$

$$\leq 1 + mb \cdot \max_{1 \leq n_r \leq m} \binom{t}{n_r} \cdot \binom{b}{n_r} \cdot 2^{b}$$

The last inequality follows due to $n_r \leq n_t \leq b$. We have $\binom{n}{k} \leq (ne/k)^k$, thus $\binom{b}{n_r} \leq (eb/n_r)^{n_r}$. Continuing the derivation we get:

$$\leq 1 + mb \cdot \max_{1\leq n_r \leq m} (et/n_r)^{n_r} \cdot (eb/n_r)^{n_r} \cdot 2^b \leq 1 + mb \cdot 2^b \max_{1\leq n_r\leq m} (e^2bt/n_r^2)^{n_r}$$

Next we compute the maximum:

$$\frac{d}{dn_r}(e^2bt/n_r^2)^{n_r} = (e^2bt/n_r^2)^{n_r} \cdot (\ln(e^2bt/n_r^2) - 2) = 0$$

$$\Leftrightarrow \ln(e^2bt/n_r^2) - 2 = 0 \Leftrightarrow e^2bt/n_r^2 = e^2 \Leftrightarrow n_r = \sqrt{bt}$$

For $m \geq \sqrt{bt}$ we get: $1 + mb \cdot 2^b \max_{1\leq n_r\leq m}(e^2bt/n_r^2)^{n_r} \leq (m+1) \cdot 2^b \cdot (e^2)^{\sqrt{bt}}$
For $m < \sqrt{bt}$: $1 + mb \cdot 2^b \max_{1\leq n_r\leq m}(e^2bt/n_r^2)^{n_r} \leq (m+1)(b+1) \cdot 2^b(e^2bt/m^2)^m$

Taking the logarithm yields the amount of transmittable information being asymptotically equal to $O(m\log(tb/m^2) + b)$ for $m < \sqrt{bt}$, since $t \geq m$ and $b \geq m$ and $O(b + \sqrt{bt}) = O(b)$ for $b \geq m \geq \sqrt{bt}$.

With the same reasoning as before a lower bound can be computed. We use $\binom{n}{k} \geq (n/k)^k$ we have $\binom{b-1}{n_r-1} \geq ((b-1)/(n_r-1))^{n_r-1}$.

$$1 + \sum_{n_r=1}^{m} \sum_{n_t=n_r}^{b} \binom{t}{n_r} \cdot \binom{n_t-1}{n_r-1} \cdot 2^{n_t} \geq \max_{1\leq n_r\leq m, 0\leq n_t\leq b} \binom{t}{n_r} \cdot \binom{n_t-1}{n_r-1} \cdot 2^{n_t}$$

$$\geq \max_{1\leq n_r\leq m} \binom{t}{n_r} \cdot \binom{b-1}{n_r-1} \cdot 2^b \geq 2^b \max_{1\leq n_r\leq m} (t/n_r)^{n_r} \cdot ((b-1)/(n_r-1))^{n_r-1}$$

$$\geq 2^b \max_{1\leq n_r\leq m} ((b-1)t/((n_r-1)n_r))^{n_r-1} \geq 2^b \max_{1\leq n_r\leq m} ((b-1)t/n_r^2)^{n_r-1}$$

Next we compute the maximum:

$$\frac{d}{dn_r}((b-1)t/n_r^2)^{n_r-1} = ((b-1)t/n_r^2)^{n_r-1} \cdot (\ln(((b-1)t/n_r^2)) - 2 + 1/n_r) = 0$$

$$\Leftrightarrow \ln((b-1)t/n_r^2) - 2 + 1/n_r = 0 \Leftrightarrow (b-1)t/n_r^2 = e^{2-1/n_r} \Leftrightarrow n_r = \sqrt{(b-1)t}/e^{1+1/(2n_r)}$$

For $m \geq \sqrt{(b-1)t}/e^{1+1/(2n_r)}$ we have: $2^b \max_{1\leq n_r\leq m}((b-1)t/n_r^2)^{n_r-1} \geq 2^b(e^2)^{\sqrt{(b-1)t}/e^2-1}$
For $m < \sqrt{(b-1)t}/e^{1+1/(2n_r)}$: $2^b \max_{1\leq n_r\leq m}((b-1)t/n_r^2)^{n_r-1} \geq 2^b(e^2(b-1)t/m^2)^{m-1}$

This, yields $\Omega(m\log(tb/m^2) + b)$ for $m < \sqrt{(b-1)t}/e^{1+1/(2n_r)}$, since $t \geq m$ and $b \geq m$ and $\Omega(b + \sqrt{bt}) \geq \Omega(b)$ for $m \geq \sqrt{(b-1)t}/e^{1+1/(2n_r)}$.

Overall, the bounds become $\Theta(m\log(tb/m^2) + b)$ for $m < \sqrt{bt}$ and $\Theta(b)$ otherwise.

Corollary 2. *The amount of information that k parties can exchange within t rounds, where each party can communicate with each other party directly and uses up to $m \leq t$ messages (at most one message per round) with a total of $b \geq m$ bits (i.e. at least one bit per message) is $\Theta(km \log(tb/m^2) + b)$ for $m < \sqrt{bt}$ and $\Theta(kb)$ bits for $m \geq \sqrt{bt}$.*

Proof. Compared to Theorem 1, where one node A only transmits data to another node B, the number of observable behaviors if k nodes are allowed to transmit is raised to the power of k, i.e. if one node can communicate in x different ways then the total number of observable behaviors becomes x^k. Taking the logarithm gives the amount of exchangeable information for k parties, i.e. $\log(x^k) = k \log x$, where $\log x$ is the amount of information a single node can transmit as stated in Theorem 1.

4.1 Lower Bound on the Time Complexity Depending on the Bit (and Message) Complexity

We first bound the amount of information that must be exchanged to solve the MIS and coloring problem. Then we give a lower bound on the time complexity depending on the bit complexity for any MIS and coloring algorithm where a message consists of at least one bit.

Theorem 3. *Any algorithm computing a MIS (in a randomized manner) in a constant degree graph, where each node can communicate less than $c_0 \log n$ bits for some constant c_0 fails (w.h.p.).*

The intuition of the so called "fooling set" argument proof is as follows: If a node cannot figure out what its neighbors are doing, i.e. it is unaware of the IDs of its neighbors, its chances to make a wrong choice are high.

Proof. Let us look at a graph being a disjoint union of cliques of size 2, i.e. every node has only one neighbor. A node can communicate in up to $2^{c_0 \log n} = n^{c_0}$ distinct ways. In the deterministic case, let $B_u \in [0, n^{c_0} - 1]$ be the behavior that a node u decides on given that it sent and received the same information throughout the execution of the algorithm. Clearly, before the first transmission no information exchange has occurred and each node u fixes some value B_u. Since there are only n^{c_0} distinct values for n nodes, there exists a behavior $B \in [0, n^{c_0} - 1]$, which is chosen by at least n^{1-2c_0} nodes given that they sent and received the same information.

Consider four arbitrary nodes $U = \{u, v, w, x\}$ that receive and transmit the same information. Consider the graph with $G' = (U, \{(u, v), (w, x)\})$ where u, v and also w, x are incident, $G'' = (U, \{(v, w), (u, x)\})$ and $G''' = (U, \{(u, w), (v, x)\})$. Note that u, v, w, x have no knowledge about the identity of their neighbors (They only know that their degrees are 1). Assume a deterministic algorithm correctly computes a MIS for G' and v is joined the MIS, then u is not joined the MIS in G' but also not in G''', since it cannot distinguish G' from G'''. Thus w must join the MIS to correctly compute a MIS for

G'''. Therefore, both v, w are joined the MIS S in G'' and thus S violates the independence condition of a MIS.

For the randomized case the argument is similar. Before the first transmission all nodes have received the same information. Since there are only n^{c_0} distinct behavior for n nodes at least a set S of nodes of cardinality $|S| \geq n^{1-c_0}$ will decide to transmit the same value B with probability at least $1/n^{2c_0}$ (given a node sent and received the same information). Now, assume we create a graph by iteratively removing two randomly chosen nodes $u, v \in S$ and adding an edge (u, v) between them until S is empty. For each node $v \in S$ must specify some probability to be joined the MIS. Assume the algorithm sets at least $|S|/2$ nodes to join with probability at least $1/2$. Given that at most $|S|/4$ nodes have been chosen from S to form pairs, the probability that two nodes u, v out of the remaining nodes joining the MIS with probability $1/2$, i.e. $\geq |S|/2 - |S|/4 = |S|/4$ nodes, are paired up is at least $1/16$ independently of which nodes have been paired up before. The probability that for a pair u, v behaving identically both nodes u, v join (and thus the computation of the MIS fails) is at least $1/4$. Since we have $|S|/2 = n^{1-2c_0}/2$ pairs, we expect a set $S' \subset S$ of at least $n^{1-6c_0}/2 \cdot 1/16 \cdot 1/4$ pairs to behave identically and join the MIS. Using a Chernoff bound for any constant $c_0 < 1/6$ at least $|S'| \geq n^{1-6c_0}/1024$ nodes behave identically with probability at least $1 - 1/n^c$ for an arbitrary constant c.

An analogous argument holds if less $|S|/2$ nodes are joined with probability more than $1/2$. In this case for some pairs u, v w.h.p. no node will join the MIS.

Theorem 4. *Any algorithm computing a MIS or coloring deterministically (or in a randomized manner) transmitting only $b \leq c_1 \frac{\log n}{\log(t/\log n)}$ bits per edge with $t \in [2\log n, n^{c_2}]$ for constants c_1, c_2 requires at least t time (w.h.p.). For $t < 2\log n$ and $b \leq c_1 \log n$ bits no algorithm can compute a MIS (w.h.p.).*[2]

Proof. If $m \geq \sqrt{tb}$ using the bound of $\Theta(b)$ of Theorem 1, a node can communicate at most $c_{thm} c_1 \log n$ bits for a constant c_{thm}. We have $c_{thm} c_1 \log n \leq c_0 \log n$ for a suitable constant c_1. Due to Theorem 3 at least $c_0 \log n$ bits are needed. For $t < 2\log n$ and $b \leq c_1 \log n$, we have $\sqrt{tb} \leq 2c_1 \log n$. If $m < \sqrt{tb}$, then the amount of transmittable information becomes (neglecting c_{thm} for now) $(m \log(tb/m^2) + b) \leq \sqrt{tb} \log 2 + c_1 \log n \leq 3c_1 \log n \leq c_0 \log n$ for a suitable constant c_1.

For $t \geq 2\log n$ and $m \leq b \leq \sqrt{tb}$ the amount of transmittable information becomes $O(m \log(tb/m^2) + b)$. We have $\max_{m \leq b} m \log(tb/m^2) \leq b \log(t/b)$. The maximum is attained for $m = b$. Using the assumption $b \leq c_1 \frac{\log n}{\log(t/\log n)}$, we get further: $b \log(t/b) \leq c_1 \frac{\log n}{\log(t/\log n)} \cdot \log(t \log(t/\log n)/\log n) = c_1 \frac{\log n}{\log(t/\log n)} \cdot (\log(t/\log n) + \log(\log(t/\log n))) = c_1 \log n \left(1 + \frac{\log(\log(t/\log n))}{\log(t/\log n)}\right) \leq 2c_1 \log n$ (since $t \leq n$). Thus, we have $m \log(tb/m^2) + b \leq 2c_1 \log n + b \leq 3c_1 \log n$.

[2] Note that the theorem does not follow directly from Theorem 3, since the number of bits that can be communicated using time-coding is generally larger than b, i.e. see Theorem 1.

Due to Theorem 3 at least $c_0 \log n$ bits required, thus for $3c_1 c_{thm} < c_0$ at least time t is required. The lower bound for the MIS also implies a lower bound for $O(\Delta)$ coloring, since a MIS for constant degree graphs can be computed from a coloring in constant time, i.e. in round i nodes with color i are joined the MIS, if no neighbor is already in the MIS.

In a later section, we give an algorithm running in time $O(t \log n)$ using $O(\log n / \log t)$ messages and bits. Thus, there exists an algorithm running in $O(\log n)$ time transmitting only $\log n/c$ messages containing one bit for any constant c. On the other hand, due to our lower bound any algorithm that transmits only one message containing $\log n/c$ bits for a sufficiently large constant c requires time n^{1/c_1} for some constant c_1 and is thus exponentially slower.

5 Algorithms Trading among Bit, Message, and Time Complexity

We look at various deterministic and randomized algorithms for the coloring and the maximal independent set problem as case studies. Before showing the tradeoffs we reduce the bit complexity of the algorithms without altering the time complexity. Then we show two mechanisms how prolonging an algorithm can be used to reduce the bit and – at the same time – the message complexity.

The first mechanism is straight forward and useful for randomized algorithms for symmetry breaking tasks, e.g. for MAC protocols where nodes try to acquire a certain resource. Assume a node tries to be distinct from its neighbors or unique among them. For example, for the coloring problem, it tries to choose a distinct color from its neighbors. For the MIS problem it tries to mark itself, and joins the MIS, if no neighbor is marked as well. Thus, if two neighbors get marked or pick the same color, we can call this a collision. We can reduce the probability of collisions by reducing the probability of a node to pick a color or get marked in a round. Thus, if a node only transmits if it has chosen a color or got marked, this causes less bits to be transmitted.

The second mechanism is beneficial for certain distributed algorithms that solve problems by iteratively solving subproblems and combining the solutions. Often the size (or number) of subproblems determines the number of iterations required, e.g. for divide and conquer algorithms. Assume that a distributed algorithm requires the same amount of communication to solve a subproblem independent of the size of the subproblem. In this case, by enlarging the size of the subproblem, the total number of iterations and thus the total amount of information to be transmitted can be reduced.

Apart from that there are also general mechanisms that work for any algorithm. Encoding information using the traditional *time coding* approach for k rounds works as follows: To transmit a value x we transmit x div k in round $x \mod k$.[3] Thus, in case $k \geq 2x - 1$ a single message of one bit is sufficient.

[3] The division operation x div k returns an integer value that states how often number k is contained in x.

Otherwise $\log x - \log k$ bits are needed. Our lower bound (Theorem 1) shows that a value of $\log x$ bits can be communicated by transmitting less than $\log x$ bits using more than one message and more than one communication round.

5.1 Coloring Algorithm

In the randomized coloring algorithms using the Multi-Trials technique [18] a node v picks a random number in $[0, \Delta]$ for each color not taken by one of its neighbors. Thus, given that a node can choose among $C(v)$ unused colors, the size of a message is $\log \Delta \cdot |C(v)|$. In [18] this is improved by letting a node pick one color out of every $\max_{u \in N(v)} 2d(u)$ colors. This results in bit complexity of $O(\log \Delta \log n)$ for $O(\Delta)$ and $\Delta + 1$ coloring. We use the improved algorithms as subroutines.

To lower the bit complexity while maintaining the same time complexity we let nodes get a color in two steps. First, a node picks an interval of colors. Second, it attempts to obtain an actual color from the chosen interval.

5.2 Randomized $O(\Delta + \log^{1+1/\log^* n} n)$ Coloring

We assume that initially each node v has $|C(v)| = (1 + 1/2^{\log^* n - 2})(\Delta_{N_+(v)} + \log^{1+1/\log^* n} n)$ colors available. Each node v considers disjoint intervals $([0, l-1], [l, 2l-1], ...)$ of colors, where each interval contains $l := (1 + 1/2^{\log^* n - 1}) \log^{1+1/\log^* n} n$ colors and the total number of intervals is given by $|C(v)|/l$. A node v first picks one of these intervals $I(v) \in \{0, 1, ..., |C(v)|/l\}$ of colors uniformly at random. From then on, it only considers a subgraph $G_{I(v)}$ of G, i.e. only neighbors $u \in N(v)$ that have picked the same interval $I(u) = I(v)$. All other neighbors operate on different intervals and have no influence on node v. Then, a coloring is computed in parallel for all subgraphs. That is to say, node v executes Algorithm *ConstDeltaColoring* [18] on $G_{I(v)}$ and tries to get a color or better said an index $ind_{I(v)}$ from $\{0, 1, ..., l-1\}$ in the interval $I(v)$. Its final color is given by the $ind_{I(v)}$ plus the color offset $I(v) \cdot l$ of the chosen interval $I(v)$.

Lemma 1. *Each node v has at most $\log^{1+1/\log^* n} n$ neighbors $u \in N(v)$ with $I(u) = I(v)$ w.h.p.*

Proof. Initially, each node picks independently uniformly at random one interval out of $(1 + 1/2^{\log^* n - 2})\Delta_{N_+(v)}/((1 + 1/2^{\log^* n - 1}) \log^{1+1/\log^* n} n) = c_1 \cdot \Delta_{N_+(v)}/\log^{1+1/\log^* n} n$ many with $c_1 = (2^{\log^* n - 1} + 2)/(2^{\log^* n - 1} + 1)$. Thus, a node v expects $E \leq \frac{\Delta_{N_+(v)}}{c_1 \cdot \Delta_{N_+(v)}/\log^{1+1/\log^* n} n} = \log^{1+1/\log^* n} n/c_1$ neighbors to have chosen the same interval. Using a Chernoff bound the probability that there are more than a factor $1 + c_1/2$ nodes beyond the expectation for a single interval is bounded by $1 - 2^{-c_1^2/8 \cdot E} = 1 - 2^{-c_1/8 \log^{1+1/\log^* n} n/c_1} \geq 1 - 1/n^{c_0}$ for an arbitrary constant c_0. Thus, w.h.p. the number of nodes in an interval is at most $(1 + c_1/2) \cdot \log^{1+1/\log^* n} n/c_1 \leq \log^{1+1/\log^* n} n$. The probability that this holds for all intervals can be bounded to be $1 - 1/n^{c_0 - 3}$ using Theorem 2 from [18].

Theorem 5. *The algorithm computes an $O(\Delta + \log^{1+1/\log^* n} n)$ coloring with bit complexity $O(\log n \log\log n)$ in time $O(\log^* n)$ w.h.p. (for sufficiently large n).*

Proof. The initial transmission of the interval requires at most $\log n$ bits, i.e. $\log \Delta - \log\log n$. Afterwards, when all nodes are split into subgraphs, the same analysis applies as for the *ConstDeltaColoring* Algorithm from [18] with $\Delta \leq \log^{1+1/\log^* n} n$, since each node only competes with at most $\log^{1+1/\log^* n} n - 1$ other nodes due to Lemma 1 and we have $(1 + 1/2^{\log^* n-1}) \log^{1+1/\log^* n} n$ available colors. The colors are picked such that the chance of getting a chosen color is constant, i.e. a node u picks one color for every sequence of $2\Delta_{N_+(v)}$ available colors, where $\Delta_{N_+(v)}$ denotes the maximum size of an uncolored neighborhood of an uncolored node $v \in N(u)$ before the current communication round. Thus, each node v that picks a color has probability $1/2$ to actually get a color independent of the choices of its neighbors, since the number of chosen colors of all neighbors together is at most $\Delta_{N_+(v)}$, i.e. half the colors of all available colors $2\Delta_{N_+(v)}$ and node v makes its choice independent of the concurrent choices of its neighbors. Thus, after a node has picked and transmitted $O(\log n)$ colors with probability $1 - 1/2^{O(\log n)} = 1 - 1/n^c$ for an arbitrary constant c, a node has obtained a color. Since each color requires $\log\log n$ bits the total bit complexity is $O(\log n \log\log n)$. We can apply Corollary 14 [18] that gives a running time of $O(\log^* n)$ w.h.p.

5.3 Rand. $O(\Delta)$ Coloring in Time t^c using $O(\log n/\log t)$ Bits

One could use the previously described algorithm and traditional time coding to save on the bit complexity maintaining the same number of transmitted messages. For readability and to illustrate both concepts quantitatively we focus on the case $t^c \geq \log^{2+\epsilon} n$ (for an arbitrary small constant ϵ), where one can save on both: the message complexity by a factor of $\log t$ and the bit complexity by a factor of $\log\log n \log t$.[4] A node initially chooses an interval consisting of $(1 + 1/2^{\log^* n-2}) \log^{1+1/\log^* n} n$ colors. Then the node iteratively transmits a single bit in a random round out of every $t_p = t^c/(c_1 \log n^{1+1/\log^* n})$ rounds. If it is the only one transmitting, it chooses a color, informs its neighbors about the obtained color and ends the algorithm.

Theorem 6. *The algorithm computes an $O(\Delta + \log^{1+1/\log^* n} n)$ coloring with bit complexity $O(\log n/\log t)$ in time $t^c + O(\log^* n)$ for any parameter c and t such that $t^c \geq \log^{2+\epsilon} n$ and $t \leq n$ for an arbitrary constant $\epsilon > 0$ w.h.p.*

Proof. The initial transmission of the interval requires less than $\log n$ bits, i.e. $\log \Delta - \log\log n$. We can use Theorem 1 with $b = m = O(\log n/\log t)$ messages m and bits b and at least $t^c/2 \geq (\log^{2+\epsilon} n)/2$ rounds. Since $\sqrt{bt} > m$ the amount of information that can be communicated is

[4] For small $t \leq 2\log n$ it is not possible to achieve bit complexity $c_1 \log n/\log t$ for a fixed constant c_1 due to the lower bound given in Theorem 4.

Algorithm FewBitsDeltaColoring, i.e. $(1+\epsilon)\Delta$ for $\epsilon > 1/2^{\log^* n-2}$ and parameter $t > \log^{2+\epsilon} n$

```
1: $s(v) := none$;  $ind_{I(v)} := none$;  $C(v) := \{0, 1, ..., (1+\epsilon)\log^{1+1/\log^* n} n - 1\}$
2: $I(v) :=$ random integer $r \in [0, (1+\epsilon)\Delta_{N_+(v)}/\log^{1+1/\log^* n} n + 1]$
3: Transmit $I(v)$ to all neighbors $u \in N(v)$ using time $t^c/2$ and $\log n/\log t$ bits and
   messages
4: $N_{I(v)}(v) := \{u \in N(v) | I(v) = I(u)\}$ {Only consider nodes in the same interval}
5: $i := 0$;  $t_p := t^c/(c_1 \log^{1+1/\log^* n} n)$ {with constant $c_1$}
6: repeat
7:    if $i \mod t_p = 0$ then $t_s(v) :=$ Random odd number in $[0, t_p]$ end if
8:    if $t_s(v) = i$ then
9:       Transmit 1
10:      if nothing received then
11:         $ind_{I(v)} :=$ arbitrary available color
12:         Transmit $ind_{I(v)}$
13:      end if
14:   end if
15:   $N(v) := \{u | u \in N_{I(v)}(v) \wedge color_I(u) = none\}$
16:   $C(v) := C(v) \setminus \{ind_I(u) | u \in N(v)\}$
17:   $i := i + 1$
18: until $ind_{I(v)} \neq none$
19: $color(v) := ind_{I(v)} + I(v) \cdot (1+\epsilon)\log^{1+1/\log^* n} n$
```

$\Theta(m\log(tb/m^2) + b) = O(\log n/\log t \log(t/(\log n/\log t)) + \log n/\log t) \geq O(\log n/\log t \log(t \log t/\log n)) \geq O(\log n/\log t(\log t + \log(\log t/\log n))) = O(\log n + \log n(\log\log t/\log t - \log\log n/\log t) = O(\log n)$ bits, since $\log\log n/\log t^c < 1/2$ because $t^c \geq \log^{2+\epsilon} n$.

Due to Lemma 1 each node v has at most $\Delta_0 := \log^{1+1/\log^* n} n$ neighbors competing for the $(1+1/2^{\log^* n-2})\log^{1+1/\log^* n} n$ colors of v's chosen interval. A node v transmits one bit for each interval of length t_p. Since nodes make their choices independently, the probability that node v is the only node transmitting is at least $1 - \Delta_0/t_p$, corresponding to the worst case that all neighbors transmit in different rounds. We have $\Delta_0/t_p = \Delta_0/(t^c/(c_2\log^{1+1/\log^* n} n) = \Delta_0 \cdot c_2 \log^{1+1/\log^* n} n/t^c \leq c_2 \log^{2+2/\log^* n} n/t^c$(due to Lemma 1) $\leq 1/t^{c \cdot 1/c_3}$ for some constant c_3 since $t^c \geq \log^{2+\epsilon} n$.Thus the chance O$(\log n/\log t) = c_4 \log n/\log t$ trials fail is $(1/t^{c/c_3})^{c_4 \log n/\log t} = 1/n^{c_1}$ for an arbitrary constant c_1 and a suitable constant c_4.

5.4 Deterministic $\Delta + 1$ Coloring

We adapt an algorithm [10] to turn a Δ^k coloring for any constant k into a $\Delta+1$ coloring in time $O(t^c\Delta\log\Delta)$ using $O(\log\Delta/\log t)$ messages of size $O(\log\Delta)$ and for an arbitrary parameter t and arbitrary constant c. The algorithm reduces the number of used colors in an iterative manner by splitting up the whole range of colors into sequences of colors of size at least $2t^c(\Delta+1)$. Consider all nodes

S_I that have a color in some sequence I consisting of $2t^c(\Delta + 1)$ colors, e.g. $I = \{0, 1, ..., 2t^c(\Delta + 1) - 1\}$. To compress the range of used colors to $[0, \Delta]$, we can sequentially go through all $2t^c(\Delta + 1)$ colors and let node $v \in S_I$ choose the smallest available color, i.e. in round i a node having the ith color in the interval I can pick a new color from I not taken by any of its neighbors. After going through all colors, we combine $2t^c$ intervals $\{I_0, I_1, ..., I_{2t^c-1}\}$ to get a new interval I' of the same size, i.e. $2t^c(\Delta + 1) - 1$. A node v with color i from I_j with $i \in [0, \Delta]$ gets color $c(v) = j \cdot (\Delta + 1) + i$ in I'. Then we (recursively) apply the procedure again on all intervals I'.

Theorem 7. *The deterministic $\Delta + 1$ coloring terminates in time $O(t^c \Delta \log \Delta)$ having bit complexity $O(\log^2 \Delta / \log t)$ and message complexity $O(\log \Delta / \log t)$ for any parameter $1 < t \leq \Delta$ and any constant c.*

Proof. We start from a correct Δ^k coloring for some constant k. A node gets to pick a color out of a sequence $(c_1, c_1 + 1, ..., c_1 + 2t^c(\Delta + 1) - 1)$ of $2t^c(\Delta + 1)$ colors for $c_1 := c_0 2t^c(\Delta + 1)$ and an arbitrary integer c_0. Thus it can always pick a color being at most $c_1 + \Delta$ since it has at most Δ neighbors. After every combination of intervals requiring time $2t^c(\Delta+1)$, the number of colors is reduced by a factor of $2t^c$. We require at most $x = k \log \Delta / \log(2t^c)$ combinations since $(2t^c)^x = \Delta^k$. Therefore, the overall time complexity is $O(t^c \Delta \log \Delta)$. In each iteration a node has to transmit one color out of $2t^c(\Delta + 1)$ many, i.e. a message of $\log(2t^c(\Delta + 1)) = O(\log \Delta)$ bits (since $t^c \leq \Delta$) giving $O(\log^2 \Delta / \log t)$ bit complexity.

5.5 MIS Algorithm

Our randomized Algorithm LowBitAndFast is a variant of algorithm [14]. It proceeds in an iterative manner. A node v marks itself with probability $1/d(v)$. In case two or more neighboring nodes are marked, the choice which of them is joined the MIS is based on their degrees, i.e. nodes with higher degree get higher priority. Since degrees change over time due to nodes becoming adjacent to nodes in the MIS, the degree has to be retransmitted whenever there is a conflict. Our algorithm improves Luby's algorithm by using the fact that the degree $d(u)$ of a neighboring node is not needed precisely, but an approximation $\tilde{d}(u)$ is sufficient. Originally, each node maintains a power of two approximation of the degrees of its neighbors, i.e. the approximation is simply the index of the highest order bit equal to 1. For example, for $d(v)$ having binary value 10110, it is 4. The initial approximate degree consists of $\log \log n$ bits. It is transmitted using (well known) time coding for $x = \log n$ rounds, i.e. to transmit a value k we transmit k div x in round $k \mod x$. When increasing the time complexity by a factor of t^{c_0} for an arbitrary constant c_0, a node marks itself with probability $1/(t^{c_0}\tilde{d}(v))$ for t^{c_0} rounds, where the approximation is only updated after the t^{c_0} rounds. Afterwards, a node only informs its neighbors if the degree changed by a factor of at least two. For updating the approximation we use time message coding for t^{c_0} rounds and a constant number of messages and bits. Whenever a

```
Algorithm LowBitAndFast for arbitrary value $t^{c_0} \geq 16$

For each node $v \in V$ :
 1: $\tilde{d}(v)$ := index of highest order bit of $2|N(v)|$ {2 approximation of $d(v)$}
 2: Transmit $\tilde{d}(v)$ to all neighbors $u \in N(v)$ using time coding for $\log n$ rounds
 3: loop
 4:    for $i = 1..t^{c_0}$ do
 5:       Choose a random bit $b(v)$, such that $b(v) = 1$ with probability $\frac{1}{4t^{c_0} \cdot \tilde{d}(v)}$
 6:       Transmit $b(v)$ to all nodes $u \in N(v)$
          if $b(v) = 1 \wedge \nexists u \in N(v), b(u) = 1 \wedge \tilde{d}(u) \geq \tilde{d}(v)$ then Join $MIS$ end if
 7:    end for
 8:    $k(v) := \max\{\lceil \log i \rceil | \text{integer } i, \frac{\tilde{d}(v)}{i} \geq d(v)\}$
 9:    if $k(v) > c_0/2 \log t$ then
10:       Transmit $k(v)$ using time message coding for $t^{c_0}$ rounds using $c_2$ messages
          of size 1 bit
11:       $\tilde{d}(v) := \tilde{d}(v) \text{ div } 2^{k(v)} + \tilde{d}(v) \mod 2^{k(v)}$
12:    end if
13:    for all received messages $k(u)$ do
14:       $\tilde{d}(u) := \tilde{d}(u) \text{ div } 2^{k(u)} + \tilde{d}(u) \mod 2^{k(u)}$
15:    end for
16: end loop
```

node is joined the MIS or has a neighbor that is joined, it ends the algorithm and informs its neighbors.

Theorem 8. *Algorithm* LowBitAndFast *terminates in time* $O(t^{c_0} \log n)$ *w.h.p. having bit and message complexity* $O(\log n / \log t)$.

The analysis is analogous to [14] and differs only in the constants. The proof can be found in [21].

References

1. Barenboim, L., Elkin, M.: Distributed $(\delta + 1)$-coloring in linear (in δ) time. In: Symposium on Theory of Computing(STOC) (2009)
2. Barenboim, L., Elkin, M.: Deterministic distributed vertex coloring in polylogarithmic time. In: Symp. on Principles of Distributed Computing (PODC) (2010)
3. Dinitz, Y., Moran, S., Rajsbaum, S.: Bit complexity of breaking and achieving symmetry in chains and rings. J. ACM (2008)
4. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed computing with advice: information sensitivity of graph coloring. In: Distributed Computing (2009)
5. Fraigniaud, P., Giakkoupis, G.: On the bit communication complexity of randomized rumor spreading. In: SPAA (2010)
6. Frederickson, G.N., Lynch, N.A.: Electing a leader in a synchronous ring. J. ACM 34(1) (1987)
7. Kothapalli, K., Scheideler, C., Onus, M., Schindelhauer, C.: Distributed coloring in $O(\sqrt{\log n})$ bit rounds. In: International Parallel & Distributed Processing Symposium, IPDPS (2006)

8. Kuhn, F.: Weak Graph Coloring: Distributed Algorithms and Applications. In: Parallelism in Algorithms and Architectures, SPAA (2009)
9. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What Cannot Be Computed Locally! In: Symposium on Principles of Distributed Computing, PODC (2005)
10. Kuhn, F., Wattenhofer, R.: On the Complexity of Distributed Graph Coloring. In: Symp. on Principles of Distributed Computing, PODC (2006)
11. Kushilevitz, E., Nisan, N.: Communication complexity. Cambridge University Press, Cambridge (1997)
12. Linial, N.: Locality in Distributed Graph Algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)
13. Lotker, Z., Patt-Shamir, B., Pettie, S.: Improved distributed approximate matching. In: SPAA (2008)
14. Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM Journal on Computing 15, 1036–1053 (1986)
15. Métivier, Y., Robson, J.M., Nasser, S.-D., Zemmar, A.: An optimal bit complexity randomized distributed MIS algorithm. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 323–337. Springer, Heidelberg (2010)
16. Nisan, N., Wigderson, A.: Rounds in communication complexity revisited. SIAM J. Comput. 22(1) (1993)
17. Santoro, N.: Design and Analysis of Distributed Algorithms. Wiley-Interscience, Hoboken (2006)
18. Schneider, J., Wattenhofer, R.: A New Technique For Distributed Symmetry Breaking. In: Symp. on Principles of Distributed Computing, PODC (2010)
19. Schneider, J., Wattenhofer, R.: Distributed Coloring Depending on the Chromatic Number or the Neighborhood Growth. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 246–257. Springer, Heidelberg (2011)
20. Schneider, J., Wattenhofer, R.: Poster Abstract: Message Position Modulation for Power Saving and Increased Bandwidth in Sensor Networks. In: 10th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN (2011)
21. Schneider, J., Wattenhofer, R.: Trading Bit, Message, and Time Complexity of Distributed Algorithms. TIK Technical Report 339 (2011), `ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-339.pdf`

Combinatorial Algorithms for Distributed Graph Coloring

Leonid Barenboim[⋆] and Michael Elkin[⋆⋆]

Department of Computer Science, Ben-Gurion University of the Negev, POB 653, Beer-Sheva 84105, Israel
{leonidba,elkinm}@cs.bgu.ac.il

Abstract. Numerous problems in Theoretical Computer Science can be solved very efficiently using powerful algebraic constructions. Computing shortest paths, constructing expanders, and proving the PCP Theorem, are just a few examples of this phenomenon. The quest for combinatorial algorithms that do not use heavy algebraic machinery, but have the same (or better) efficiency has become a central field of study in this area. Combinatorial algorithms are often simpler than their algebraic counterparts. Moreover, in many cases, combinatorial algorithms and proofs provide additional understanding of studied problems. In this paper we initiate the study of combinatorial algorithms for Distributed Graph Coloring problems. In a distributed setting a communication network is modeled by a graph $G = (V, E)$ of maximum degree Δ. The vertices of G host the processors, and communication is performed over the edges of G. The goal of distributed vertex coloring is to color V with $(\Delta + 1)$ colors such that any two neighbors are colored with distinct colors. Currently, efficient algorithms for vertex coloring that require $O(\Delta + \log^* n)$ time are based on the algebraic algorithm of Linial [18] that employs set-systems. The best currently-known combinatorial set-system free algorithm, due to Goldberg, Plotkin, and Shannon [14], requires $O(\Delta^2 + \log^* n)$ time. We significantly improve over this by devising a *combinatorial* $(\Delta + 1)$-coloring algorithm that runs in $O(\Delta + \log^* n)$ time. This exactly matches the running time of the best-known *algebraic* algorithm. In addition, we devise a tradeoff for computing $O(\Delta \cdot t)$-coloring in $O(\Delta/t + \log^* n)$ time, for almost the entire range $1 < t < \Delta$. We also compute a Maximal Independent Set in $O(\Delta + \log^* n)$ time on general graphs, and in $O(\log n/ \log\log n)$ time on graphs of bounded arboricity. Prior to our work, these results could be only achieved using algebraic techniques. We believe that our algorithms are more suitable for real-life networks with limited resources, such as sensor, ad-hoc, and mobile networks.

[⋆] Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

[⋆⋆] This research has been supported by the Binational Science Foundation, grant No. 2008390. Additional funding was provided by the Lynne and William Frankel Center for Computer Sciences.

1 Introduction

1.1 Algebraic versus Combinatorial Algorithms

It is a common scenario in Theoretic Computer Science that very strong results are achieved by powerful non-combinatorial techniques. In many occasions consequent research focuses on devising *combinatorial* counterparts to these results. This quest for combinatorial algorithms is often justified by the desire to obtain better understanding of the problem at hand. In some cases it also leads to more efficient and simple algorithms.

One notable example of such development is the celebrated *PCP theorem*. This famous result was achieved [3,12] by algebraic techniques. Recently a significant research effort was invested in devising a combinatorial proof for this result [10,21]. Another important example is *expanders* and *Ramanujan* graphs. Near-optimal algebraic constructions of expanders were devised in [1,19]. Reingold, Vadhan and Wigderson [26] devised the first *combinatorial* construction of expanders. Though the parameters of these combinatorial constructions are somewhat inferior to algebraic ones, the techniques that were developed as a part of this effort turned out to be useful for devising a log-space S-T-connectivity algorithm [25]. Also, consequently to the work of [26], improved combinatorial constructions of expanders and near-Ramanujan graphs were devised in [8].

This phenomenon occurs also in *approximation* algorithms. Linear and semidefinite programming is an extremely powerful algebraic technique in this area. However, it is an active line of research to explore how well one can do *without linear programming*. Yet another example of this phenomenon is algorithms for computing (almost) *shortest paths*. Very efficient algorithms for this problem were achieved about twenty years ago via fast matrix multiplication. Recently, combinatorial algorithms for this problem were devised, which, in some scenarios, outperform their algebric counterparts.

1.2 Distributed Coloring

We study the *distributed coloring* problem. We are given an n-vertex unweighted undirected graph $G = (V, E)$, with each vertex $v \in V$ hosting a processor. The processors share no common memory. They communicate with each other by sending short messages (of size $O(\log n)$ each) over the edges of E. The communication is synchronous, i.e., it occurs in discrete rounds. All vertices wake up simultaneously. Each vertex $v \in V$ has a unique identity number ($Id(v)$). For simplicity we assume that all identifiers are from the range $\{1, 2, ..., n\}$. All algorithms extend to larger ranges of identifiers.

Denote by Δ the maximum degree of G. In the $(\Delta + 1)$-*coloring* problem the objective is to color G with $\Delta + 1$ colors *legally*, i.e., in such a way that for every edge $e = (u, v)$, the endpoints u and v will get distinct colors. The *running time* of an algorithm is the number of rounds that elapse until all vertices compute their final colors. Another closely related problem is the Maximal Independent Set (henceforth, MIS) problem. In this problem we want to compute a subset $U \subseteq V$ of independent vertices (i.e., for every $u, u' \in U$, there is no edge (u, u')

in the graph) with the property that for every vertex $v \in V \setminus U$, there exists a neighbor $u \in U$ of v.

These two problems are widely considered to be among the most fundamental distributed problems. Recently, a significant progress was achieved in devising deterministic distributed algorithms for them. Specifically, the authors of the current paper [6], and independently Kuhn [16], devised a $(\Delta + 1)$-coloring algorithm with running time $O(\Delta + \log^* n)$. These algorithms also directly give rise to algorithms that compute MIS within the same time. Both papers [6,16] also devised a tradeoff, and showed that for any constant ϵ, $0 < \epsilon < 1$, a $\Delta^{1+\epsilon}$-coloring can be computed in $O(\Delta^{1-\epsilon} + \log^* n)$ time. (The results in [16] are, in fact, even more general than this; they show that for any parameter t, $1 \le t \le \Delta$, a Δ/t-coloring can be computed in $O(\Delta \cdot t + \log^* n)$ time.) Finally, on graphs of small *arboricity* [1], the authors of the current paper devised in [5] an algorithm that computes $(\Delta + 1)$-coloring and MIS in $O(\frac{\log n}{\log \log n})$ time.

All these results rely heavily on an algorithm of Linial [18], that computes an $O(\Delta^2)$-coloring within $\log^* n + O(1)$ time. The latter seminal result relies, in turn, on an algebraic construction of Erdős, Frankl and Füredi [11] of set-systems with certain special and very useful properties. Moreover, the algorithm of Kuhn [16] takes this algebraic technique one step further, and devises an algebraic construction of sets of functions that are tailored for the needs of his coloring algorithm. We remark also that even previous to the work of [6,16] $(\Delta + 1)$-coloring algorithms relied on algebraic techniques. Specifically, the $(\Delta + 1)$-coloring and MIS algorithms of Kuhn and Wattenhofer [17] that require $O(\Delta \log \Delta + \log^* n)$ time rely on Linial's algorithm. To the best of our knowledge, the best currently known deterministic algorithm of this type that does not rely on Linial's method is the algorithm due to Goldberg, Plotkin and Shannon [14]. The latter algorithm requires, however, $O(\Delta^2 + \log^* n)$ time.

The basic question that we investigate in the current paper is whether algebraic techniques are indeed necessary for devising efficient deterministic coloring and MIS algorithms. We demonstrate that it is not the case, and devise *combinatorial* (we also call them *set-system free*) coloring and MIS algorithms whose performance matches the state-of-the-art. Specifically, one of our new combinatorial algorithms computes a $(\Delta+1)$-coloring and MIS within $O(\Delta+\log^* n)$ time, Another one provides a tradeoff and computes a $\Delta^{1+\epsilon}$-coloring in $O(\Delta^{1-\epsilon} + \log^* n)$ time, for any constant ϵ, $0 < \epsilon < 1$. We also devise combinatorial $(\Delta+1)$-coloring and MIS algorithms for graphs of small arboricity that run in $O(\frac{\log n}{\log \log n})$ time.

We believe that the value of these results is two-fold. First, it addresses the aforementioned question, and shows that like in the context of PCP and expanders, one also can get rid of algebraic techniques in the context of distributed deterministic coloring. By this our algorithms seem to uncover a new understanding of the nature of the explored problems. Second, since our algorithms are combinatorial, they are much easier for implementation by not-very-mathematically-inclined programmers. In addition, the combinatorial nature of our algorithms enables for more efficient implementation in terms of local computation. While the latter is

[1] See Section 2 for its definition.

usually suppressed when analyzing distributed algorithms, it may become crucial when running the algorithms on certain simple real-world communication devices, such as sensors or antennas.

1.3 Our Techniques

The most tempting approach to the problem of devising combinatorial coloring algorithms is to devise a combinatorial counterpart to Linial's algorithm. However, we were not able to accomplish this. Instead we observe that some of the aforementioned coloring algorithms [5,6] start with computing an $O(\Delta^2)$-coloring via Linial's algorithm, and then employ this coloring in a way that can be relatively easily made combinatorial. Our new algorithms invoke neither the algorithm of Linial itself, nor a combinatorial analogue of it. Instead, in our algorithms we start with partitioning the edge set of the graph into Δ forests (using Panconesi-Rizzi algorithm [22]). We then color each forest separately, and combine colorings of pairs of forest. In this way we obtain an $O(\Delta^2)$-coloring of each of the $\Delta/2$ pairs of forests. Next, we employ combinatorial algorithms to manipulate the colorings in each of these pairs of forests, and to obtain, rougly speaking, a $(\Delta+1)$-coloring for each pair. We then merge these pairs again, and manipulate the resulting coloring again. We iterate in this way up until we get a unified coloring for the entire graph. Obviously, this schematic outline suppresses many technical details. Also, this scheme does not achieve our best results, which we obtain by using more sophisticated combinatorial methods. Nevertheless, it seems to capture the basic ideas behind our combinatorial algorithms.

1.4 Related Work

Goldberg et al. [13] (based on [9]) devised $(\Delta + 1)$-coloring and MIS algorithms that require $O(\Delta^2 + \log^* n)$ time. Linial [18] stregthened this result, and showed that an $O(\Delta^2)$-coloring can be computed in $\log^* n + O(1)$ time. Kuhn and Wattenhofer [17] improved the running time of [13] to $O(\Delta \log \Delta + \log^* n)$. The latter was further improved to $O(\Delta + \log^* n)$ in [6,16]. On graphs of arboricity $a \leq \log^{1/2-\epsilon} n$, for a constant $\epsilon > 0$, [5] devised $(\Delta + 1)$-coloring and MIS algorithms that require $O(\frac{\log n}{\log \log n})$ time. A variety of additional algorithms and tradeoffs for coloring graphs of small arboricity were devised in [5,7].

Awerbuch, Goldberg, Luby and Plotkin [4] devised deterministic $(\Delta + 1)$-coloring and MIS algorithms that require $2^{O(\sqrt{\log n \log \log n})}$ time. The latter was improved to $2^{O(\sqrt{\log n})}$ by Panconesi and Srinivasan [23]. Randomized algorithms with logarithmic running time were devised by Luby [20], and by Alon, Babai, and Itai [2]. Kothapalli et al. [15] showed that an $O(\Delta)$-coloring can be computed in $O(\sqrt{\log n})$ randomized time. Recently, Schneider and Wattenhofer [27] showed that $(\Delta + 1)$-coloring can be computed in randomized $O(\log \Delta + \sqrt{\log n})$ time. They have also showed a tradeoff between the number of colors and running time with very efficient algorithms for $O(\Delta + \log n)$-coloring.

2 Preliminaries

2.1 Definitions and Notation

Unless the base value is specified, all logarithms in this paper are to base 2. The *degree* of a vertex v in a graph $G = (V, E)$, denoted *deg(v)*, is the number of edges incident to v. A vertex u such that $(u, v) \in E$ is called a *neighbor* of v in G. The maximum degree of a vertex in G, denoted $\Delta = \Delta(G)$, is defined by $\Delta(G) = \max_{v \in V} deg(v)$. A *forest* is an acyclic subgraph. A *tree* is a connected acyclic subgraph. A forest $\mathcal{F}$ can also be represented as a collection of vertex-disjoint tress $\mathcal{F} = \{T_1, T_2, ..., T_k\}$. A tree T is said to be oriented if (1) there is a designated *root* vertex $rt \in V(T)$, (2) every vertex $v \in V(T)$ knows whether v is the root or not, and, in the latter case, v knows which of its neighbors is the parent $\pi(v)$ of v. (The parent $\pi(v)$ of v is the unique neighbor of v that lies on the (unique) path in T connecting v with rt.) A forest $\mathcal{F} = \{T_1, T_2, ..., T_k\}$ is said to be *oriented* if each of the trees $T_1, T_2, ..., T_k$ is oriented. A *Forest-Decomposition* of a graph $G = (V, E)$ is an edge partition such that each subgraph forms an oriented forest. The *arboricity* of a graph G is the minimal number a such that the edge set of G can be covered with at most a edge disjoint forests.

A mapping $\varphi : V \rightarrow \mathbb{N}$ is called a *coloring*. A coloring that satisfies $\varphi(v) \neq \varphi(u)$ for each edge $(u, v) \in E$ is called a *legal coloring*. For a positive integer k, a k-coloring φ is a legal coloring that employs at most k colors, i.e., for each vertex v, $\varphi(v) \in \{1, 2, ..., k\}$.

For two positive integers m and p, an *m-defective p-coloring* of a graph G is a (not necessarily legal) coloring of the vertices of G using p colors, such that each vertex has at most m neighbors colored by its color. Note that each color class in an m-defective coloring induces a graph of maximum degree m.

2.2 Coloring Procedures

In this section we summarize several well-known results that are used in the current paper. Some of our algorithms use as a black-box a procedure due to Kuhn and Wattenhofer [17]. This procedure accepts as input a graph G with maximum degree Δ, and an initial legal m-coloring, and it produces a $(\Delta + 1)$-coloring of G within time $(\Delta + 1) \cdot \lceil \log(m/(\Delta + 1)) \rceil = O(\Delta \cdot \log(m/\Delta))$. We will refer to this procedure as *KW iterative procedure*, or *KW Procedure*. For future reference we summarize this in the next lemma.

Lemma 2.1. *[17] Given a legal m-coloring of a graph with maximum degree Δ, KW procedure produces a $(\Delta + 1)$-coloring within $O(\Delta \cdot \log(m/\Delta))$ time.*

We also use a Δ-forest-decomposition procedure due to Panconesi and Rizzi [22]. (A similar construction was used also by Goldberg et al. [14].) This procedure accepts as input a graph G and computes a forest-decomposition with at most Δ forests in $O(1)$ time. We will refer to this procedure as PR *Δ-Forest-Decomposition* Procedure, or *PR Procedure*.

Next, we summarize the properties of several additional procedures that are used by our algorithms. In these procedures, as well as in our algorithms, we use the following naming conventions. If a procedure involves set-systems and requires a legal coloring as input, then the suffix -SL is added to its name. If the procedure involves set-systems, but does not require a coloring as input, then the suffix -SET is added to its name. If the procedure is set-system free, but it requires a legal coloring as input, then the suffix -LEG is added to its name. The next lemma summarizes the properties of procedures Delta-Col-SL and Trade-Col-SL for computing legal colorings, and Procedure Defective-Col-LEG for computing defective colorings, devised in [6].

Lemma 2.2. *[6] **(1)** Procedure Delta-Col-SL invoked on an input graph G with an initial $O(\Delta^2)$-coloring computes a $(\Delta+1)$-coloring in $O(\Delta)$ time.*
***(2)** Given a graph G with an $O(\Delta^2)$ coloring, and an arbitrary parameter t, $1 < t \leq \Delta^{1/4}$, Procedure Trade-Col-SL computes an $O(\Delta \cdot t)$-coloring in time $O(\Delta/t)$.*
***(3)** Procedure Defective-Col-LEG accepts as input a graph G with an initial $(c' \cdot \Delta^k)$-coloring, for some constants $c' > 0$ and $k \geq 2$, and two parameters p, q such that $0 < p^2 < q$. It computes a $(\frac{\log(c' \cdot \Delta^k)}{\log(q/p^2)} \cdot \Delta/p)$-defective p^2-coloring in time $(\frac{\log(c' \cdot \Delta^k)}{\log(q/p^2)}) \cdot O(q)$. Moreover, Procedure Defective-Col-LEG is set-system-free.*

3 The Generic Method

Distributed computation of a $(\Delta+1)$-coloring in general graphs is a challenging task. However, for certain graph families very efficient, and even optimal, algorithms are known. In particular, the algorithm of Goldberg and Plotkin [13] [1] is applicable for oriented forests. (Henceforth, the GP algorithm.) The GP algorithm computes a 3-coloring of a forest in $O(\log^* n)$ time. Using PR Procedure the edge set of any graph can be partitioned into Δ oriented forests. Our algorithms start by partitioning a graph into Δ forests, and computing a 3-coloring in each forest, in parallel. Then these colorings are efficiently merged into a single unified $(\Delta+1)$-coloring.

We begin with describing a procedure called *Procedure Pair-Merge* that combines the colorings of two edge-disjoint subgraphs. Procedure Pair-Merge accepts as input a graph $G = (V, E)$, such that E is partitioned into two edge-disjoint subsets E' and E''. It also accepts two legal colorings φ' and φ'' for the graphs $G' = (V, E')$ and $G'' = (V, E'')$, respectively. The colorings φ' and φ'' employ at most c' and c'' colors, respectively. Procedure Pair-Merge returns a new coloring φ for G such that for every $v \in V$, $\varphi(v) = c'' \cdot (\varphi'(v) - 1) + \varphi''(v)$. The new coloring can be seen as an ordered pair $< \varphi'(v), \varphi''(v) >$. This completes the description of the procedure. Observe that invoking Procedure Pair-Merge requires

[1] The algorithm of [13] is based on an earlier algorithm of Cole and Vishkin [9].

no communication whatsoever. The properties of the procedure are summarized in the next lemma.

Lemma 3.1. *Procedure Pair-Merge produces a legal* $(c' \cdot c'')$*-coloring of* G.

Next, we describe a generic method [2], called Generic-Merge, for merging the coloring of ℓ edge disjoint subgraphs of G, for a positive integer ℓ. Suppose that we are given an edge partition $E_1, E_2, ..., E_\ell$ of E. Moreover, for $1 \leq i \leq \ell$, we are given a legal coloring φ_i of $G_i = (V, E_i)$ that employs at most c_i colors, $c_i \geq \Delta + 1$. In addition, the method Generic-Merge employs an auxilary coloring procedure called Procedure Reduce(H, α, β). We will later use the method Generic-Merge with a number of different instantiations of Procedure Reduce. However, in all its instantiations Procedure Reduce accepts as input a graph H with a legal α-coloring, and computes a legal β-coloring of H. Procedure Reduce requires that $\alpha > \beta \geq \Delta + 1$. The method Generic-Merge also accepts as input a positive integer parameter d, $1 \leq d \leq \min\{c_i | 1 \leq i \leq \ell\}$. Roughly speaking, the parameter d determines the ratio between α and β. To summarize, the method Generic-Merge accepts as input a partition $E_1, E_2, ..., E_\ell$ of E, a legal c_i-coloring φ_i of E_i, for each $i = 1, 2, ..., \ell$, a procedure Reduce, and a parameter d. It returns a legal coloring φ of the entire input graph G.

The method Generic-Merge proceeds in phases. In each phase pairs of subgraphs are merged using Procedure Pair-Merge, in parallel. As a result we obtain fewer subgraphs, but a greater number of colors is employed in each subgraph. The number of colors is then reduced using Procedure Reduce, which is invoked in parallel on the merged subgraphs. This process of pairing subgraphs, merging their colorings, and reducing the number of employed colors is repeated for $\lceil \log \ell \rceil$ phases, until all subgraphs are merged into the original input graph G.

Algorithm 1. Method Generic-Merge($G_1, G_2, ..., G_\Delta, \varphi_1, \varphi_2, ..., \varphi_\Delta, c_1, c_2, ..., c_\Delta$, Procedure Reduce, d)

```
1: $\ell := \Delta$   /* $\ell$ is the number of subgraphs */
2: **while** $\ell > 1$ **do**
3:    **for** $i := 1, 2, ..., \lfloor \ell/2 \rfloor$, in parallel **do**
4:       $\{G'_i, \varphi'_i\}$ := Pair-Merge($G_{2i-1}, G_{2i}, \varphi_{2i-1}, \varphi_{2i}, c_{2i-1}, c_{2i}$)
5:       $\{G_i, \varphi_i\}$ := Reduce($G'_i$,  $\alpha_i := c_{2i-1} \cdot c_{2i}$,  $\beta_i := \lfloor \alpha_i / d \rfloor$)
6:       $c_i := \lfloor c_{2i-1} \cdot c_{2i} / d \rfloor$
7:    **end for**
8:    **if** $\ell$ is odd **then**
9:       $\{G_{\lceil \ell/2 \rceil}, \varphi_{\lceil \ell/2 \rceil}, c_{\lceil \ell/2 \rceil}\} := \{G_\ell, \varphi_\ell, c_\ell\}$
10:   **end if**
11:   $\ell := \lceil \ell/2 \rceil$
12: **end while**
13: return $\{G_1, \varphi_1, c_1\}$
```

[2] We refer to a procedure as *generic method* if it accepts another procedure as input.

In the first phase of Generic-Merge the pairs (G_1, G_2), (G_3, G_4),...,$(G_{\ell-1}, G_\ell)$ are merged into the subgraphs $G'_1, G'_2, ..., G'_{\lceil \ell/2 \rceil}$ by using Procedure Pair-Merge. (In other words, $G'_i = (V, E_{2i-1} \bigcup E_{2i})$, for $i = 1, 2, ..., \lfloor \ell/2 \rfloor$. If ℓ is odd then we define $G_{\ell+1} = (V, \emptyset)$, $c_{\ell+1} = 1$, and, consequently, $G'_{\lceil \ell/2 \rceil} = G_\ell$.) Set $\alpha_i = c'_i = c_{2i-1} \cdot c_{2i}$, and $\beta_i = \lfloor \alpha_i/d \rfloor$. (Intuitively, α_i is an upper bound on the number of colors used by the coloring that Procedure Pair-Merge produces for the subgraph G'_i. Procedure Reduce transforms this coloring into a β_i-coloring, with $\beta_i = \lfloor \alpha_i/d \rfloor$.) Next, Procedure Reduce$(G'_i, \alpha_i, \beta_i)$ is executed in parallel, for $i = 1, 2, ..., \lfloor \ell/2 \rfloor$. In general, a phase proceeds as follows. Suppose that the previous phase has produced the subgraphs $G'_1, G'_2, ..., G'_{\ell'}$, such that each subgraph is colored with at most $c'_1, c'_2, ..., c'_{\ell'}$ colors, respectively. Then the subgraphs G'_{2i-1} and G'_{2i} are merged into the subgraph G''_i, and Procedure Reduce$(G''_i,\ \alpha'_i = c'_{2i-1} \cdot c'_{2i},\ \beta'_i = \lfloor c'_{2i-1} \cdot c'_{2i}/d \rfloor)$ is executed in parallel, for $i = 1, 2, ..., \lfloor \ell'/2 \rfloor$. If ℓ is odd then $G''_{\lceil \ell'/2 \rceil} = G'_{\ell'}$. In this case the coloring of G'_ℓ is also used for $G''_{\lceil \ell'/2 \rceil}$, instead of invoking Procedure Reduce on it. The method Generic-Merge terminates once all subgraphs are merged into a single graph, that is, the input graph G. This completes the description of the method Generic-Merge.

Lemma 3.2. *The method Generic-Merge produces a legal* $(c_1 \cdot c_2 \cdot .... \cdot c_\ell)/d^{\ \ell-1}$ *coloring* φ *of the input graph* G*.*

4 Coloring Algorithms

4.1 Procedure Simple-Col

In this section we present our first algorithm that employs the method Generic-Merge, called *Procedure Simple-Col.* The method Generic-Merge accepts as input a partition of the input graph such that each subgraph in the partition is legally colored. In order to compute such a partition, we invoke the PR Procedure. Recall that this procedure computes a Δ-forest-decomposition of G. In other words, the procedure outputs an edge partition $\{G_1, G_2, ..., G_\Delta\}$, $G_i = (V, E_i), i \in \{1, 2, ..., \Delta\}$, such that each subgraph in this partition is an oriented forest. Next, each forest is colored with 3 colors using the GP algorithm. The Δ invocations of the GP algorithm are performed in parallel. They result in legal colorings $\varphi_1, \varphi_2, ..., \varphi_\Delta$. Since $\Delta \geq 2$, each coloring φ_i, $i = 1, 2, ..., \Delta$, employs at most $\Delta + 1$ colors.

Recall that the method Generic-Merge also accepts as input parameter a procedure (Procedure Reduce) for reducing the number of colors in a given coloring of a graph. In Procedure Simple-Col we employ the KW iterative procedure as Procedure Reduce. (The KW iterative procedure accepts as input a graph H with an α-coloring, $\alpha \geq \Delta + 1$, and computes a $(\Delta + 1)$-coloring of H in time $O(\Delta \log(\alpha/\Delta))$.) We invoke Generic-Merge on the partition $\{G_1, G_2, ..., G_\Delta\}$ with the colorings $\varphi_1, \varphi_2, ..., \varphi_\Delta$ such that $c_1 = c_2 = ... = c_\Delta = \Delta + 1$. Finally, the parameter d is set to $\Delta + 1$. Consequently, in each phase of Generic-Merge pairs of $(\Delta + 1)$-colored subgraphs are merged into $(\Delta + 1)^2$-colored subgraphs, and then the number of colors in each subgraph is reduced back to $\Delta + 1$. Once

Generic-Merge terminates, the input graph is legally colored using $\Delta + 1$ colors. The pseudocode of the procedure follows. Its properties are summarized below.

Algorithm 2. Procedure Simple-Col(G)

```
1: $\{G_1, G_2, ..., G_\Delta\} := \Delta$-Forests-Decomposition($G$)    /*using PR Procedure */
2: for $i = 1, 2, ..., \Delta$ in parallel do
3:     $\varphi_i :=$ 3-color($G_i$)    /*using GP algorithm */
4:     $c_i := \Delta + 1$
5: end for
6: Generic-Merge($G_1, G_2, ..., G_\Delta, \varphi_1, \varphi_2, ..., \varphi_\Delta, c_1, c_2, ..., c_\Delta$, KW procedure, $\Delta + 1$)
```

Theorem 4.1. *Procedure Simple-Col produces a legal $(\Delta+1)$-coloring of G. Its running time is $O(\Delta \log^2 \Delta) + \log^* n$. This procedure is set-system free.*

4.2 Procedures Poly-Col and Fast-Col

Our algorithm consists of two major stages. In the first stage we compute a $\Delta^{O(1)}$-coloring, and in the second stage we reduce the number of colors from $\Delta^{O(1)}$ to $(\Delta+1)$. In the existing $(\Delta+1)$-coloring algorithms that run in $O(\Delta) + \log^* n$ time [6,16], both these stages employ set systems. In fact, as far as we know, currently there is no known set-system free $\Delta^{O(1)}$-coloring algorithm that runs within $O(\Delta) + \log^* n$ time. The situation is somewhat better in the context of the second stage, as there is a known set-system free algorithm (KW Procedure) that accepts a $\Delta^{O(1)}$-coloring as input and returns a $(\Delta + 1)$-coloring. However, its running time $(O(\Delta \log \Delta))$ is higher than the desired bound of $O(\Delta) + \log^* n$. Therefore, we speed up both the first and the second stages of the aforementioned scheme, and achieve a set-system free $(\Delta + 1)$-coloring algorithm with running time $O(\Delta + \log^* n)$.

Before we begin with the description of our new algorithm, we provide a brief survey of several known (not set-system free) algorithms (due to [6]) that employ the two-stage technique described above. In the sequel, we modify these algorithms, and eliminate the steps that employ set-systmes. Then we employ the modified versions for devising our new results. We start with sketching Procedure Defective-Col-SET from [6], that accepts as input a graph G and two parameters p and q, and returns a defective coloring of G.

Algorithm 3. Procedure Defective-Col-SET (G, p, q)

```
1: $\vartheta :=$ an $O(\Delta^2)$-coloring of $G$    /* using set-systems */
2: $\psi :=$ Defective-Col-LEG ($G$, $\vartheta$, $p$, $q$)    /* set-system free */
3: return $\psi$
```

Set $p = \Delta^{\epsilon}$, $q = \Delta^{3\epsilon}$, for an arbitrarily small constant $\epsilon > 0$. By Lemma 2.2 (3), Procedure Defective-Col-SET invoked with these parameters computes an $O(\Delta/p)$-defective p^2-coloring of G.

Next we sketch a procedure devised in [6] for computing a legal $O(\Delta^{5/4})$-coloring from a legal $O(\Delta^2)$-coloring in $O(\Delta^{3/4})$ time. This procedure is called Procedure Trade-Col-SL. (The properties of this procedure in its general form are summarized in Lemma 2.2 (2). In the current discussion we fix the parameter t to be equal to $\Delta^{1/4}$.) Procedure Trade-Col-SL accepts as input a graph G and a legal $O(\Delta^2)$-coloring ϑ of G. The procedure proceeds as follows. First, it computes an $O(\Delta^{3/4})$-defective $O(\Delta^{1/2})$-coloring of the input graph using Procedure Defective-Col-LEG. (See Lemma 2.2 (3).) To this end we set $p = \Delta^{1/4}$, $q = \Delta^{3/4}$. (Normally, it is preceded by a step in which an $O(\Delta^2)$-coloring is computed via Linial's algorithm, i.e., using set systems. In the current version of the procedure, this coloring is assumed to be provided as a part of the input.) The defective coloring returned by the invocation of Procedure Defective-Col-LEG induces a *vertex* partition into $O(\Delta^{1/2})$ subgraphs such that each subgraph has maximum degree $O(\Delta^{3/4})$. Next, all subgraphs are legally colored with distinct palettes of size $O(\Delta^{3/4})$ for each subgraph, in parallel. Then these colorings are combined into a unified legal $O(\Delta^{5/4})$-coloring. This completes the description of Procedure Trade-Col-SL. Similarly to Algorithm 3, the computation is divided into stages that either involve set-systems or are set-systems free.

Algorithm 4. Procedure Trade-Col-SL (G , ϑ)

```
1: $\psi$ := Defective-Col-LEG ($G$, $\vartheta$, $p := \Delta^{1/4}$, $q := \Delta^{3/4}$)   /* set-system free; see
   Lemma 2.2 (3) */
   /* $\psi$ is an $O(\Delta^{3/4})$-defective $O(\Delta^{1/2})$-coloring of $G$ */
   /* $\psi$ induces a vertex partition into $O(\Delta^{1/2})$ subgraphs, each with max. degree
   $O(\Delta^{3/4})$ */
2: for each subgraph $G_i$ induced by color classes of $\psi$, in parallel do
3:    $\varphi_i$ := color $G_i$ with $O(\Delta^{3/2})$ colors using Linial alg.[18]   /* using set-systems */
4:    $\varphi'_i$ := Delta-Col-SL($G_i$, $\varphi_i$)   /* using set-systems; see Lemma 2.2 (1) */
      /* $\varphi'_i$ is an $O(\Delta^{3/4})$-coloring of $G_i$ */
5: end for
6: for $i = 1, 2, ...$ in parallel do
7:    combine all colorings $\varphi'_i$ into a unified legal $O(\Delta^{5/4})$-coloring $\varphi$ of $G$
      /*set-system free */
8: end for
9: return $\varphi$
```

Next, we turn to describing our new set-system free algorithm for computing an $O(\Delta^{5/4})$-coloring from scratch. Our algorithm employs the method Generic-Merge, that was described in Section 3. In Section 4.1 the KW Procedure was used as an instantiation for Procedure Reduce in the method Generic-Merge. This time a different procedure is used as an instantiation for Procedure Reduce. This procedure reduces the number of colors from $c^2 \cdot \Delta^{5/2}$ to $c \cdot \Delta^{5/4}$, for a positive constants c. Its running time is $O(\Delta^{3/4} \log \Delta) = o(\Delta / \log \Delta)$. Consequently, the $\lceil \log \Delta \rceil$ phases of Generic-Merge require overall time of $o(\Delta)$. For $i = 1, 2, .., \lceil \log \Delta \rceil$, the colorings of subgraphs in the partition of phase i employ at most $(c \cdot \Delta^{5/4})$ colors each. In phase i, pairs of $(c \cdot \Delta^{5/4})$-colored subgraphs are merged into $(c^2 \cdot \Delta^{5/2})$-colored subgraphs, and the number of colors is then reduced back to $(c \cdot \Delta^{5/4})$ in each subgraph. Once the method Generic-Merge terminates, the input graph is colored with $(c \cdot \Delta^{5/4})$ colors.

We use a modified version of Procedure Trade-Col-SL (Algorithm 4), as an instantiation for Procedure Reduce. Given an $O(\Delta^2)$-coloring as input, Procedure Trade-Col-SL computes an $O(\Delta^{5/4})$-coloring in $O(\Delta^{3/4})$ time. Some of its steps employ set-systems. (See Algorithm 4.) Consequently, the original Procedure Trade-Col-SL cannot be used. We perform two modifications in the procedure. First, the steps that employ set-systems are replaced by analogous set-system free steps. Second, we show that the procedure computes an $O(\Delta^{5/4})$-coloring from an $O(\Delta^k)$-coloring for any constant $k \geq 2$, not only $k = 2$. We henceforth refer to the modified procedure as *Procedure Mod-Trade-LEG*.

Procedure Trade-Col-SL employs set-systems for computing $O(\Delta^{3/2})$-colorings of subgraphs G_i (line 3 of Algorithm 4). In addition, it employs set-systems in the invocation of Procedure Delta-Col-SL for computing legal colorings of subgraphs with linear number of colors (and in linear time) in the degree of the subgraphs (line 4 of Algorithm 4). In Procedure Mod-Trade-LEG we omit the step of computing $O(\Delta^{3/2})$-coloring of subgraphs G_i. Instead of this step, φ_i is set as the initial coloring ϑ, for all i. For each $i = 1, 2, ..., O(\Delta^{1/2})$, let $\Delta_i = \Theta(\Delta^{3/4})$ be an upper bound on the maximum degree $\Delta(G_i)$ of the graph G_i. Since ϑ is an $O(\Delta^2)$-coloring, it is also an $O(\Delta_i^{8/3})$-coloring of G_i. Next, we replace the step of coloring G_i using Procedure Delta-Col-SL with an invocation of the KW iterative procedure, which is set-systems free. This procedure can start (see Lemma 2.1) with an arbitrarily large number of colors, as long as it is polynomial in the maximum degree of the underlying graph. However, as a result, the running time of procedure Mod-Trade-LEG grows by a multiplicative factor of $\log \Delta$, and becomes $O(\Delta^{3/4} \log \Delta)$.

The original Procedure Trade-Col-SL accepts as input a $(c' \cdot \Delta^2)$-coloring ϑ, for a positive constant c', and reduces it into an $O(\Delta^{5/4})$-coloring. Once line 3 of Algorithm 4 is skipped[1], the coloring ϑ is used only in one step of Procedure Trade-Col-SL, specifically, in the step that computes the $O(\Delta^{3/4})$-defective $O(\Delta^{1/2})$-coloring (line 1 of Algorithm 4). In this step a procedure called Procedure Defective-Col-LEG is invoked with two input parameters $p = \Delta^{1/4}$ and $q = \Delta^{\epsilon} \cdot p^2$, for an arbitrary small positive constant $\epsilon \leq 1/4$. Procedure Defective-Col-LEG employs the coloring ϑ to compute a $(\frac{\log(c' \cdot \Delta^2)}{\log(q/p^2)} \cdot \Delta/p)$-defective p^2-coloring in time $(\frac{\log(c' \cdot \Delta^2)}{\log(q/p^2)}) \cdot O(q)$. Procedure Defective-Col-LEG can employ any legal t-coloring φ' instead of the $(c' \cdot \Delta^2)$-coloring ϑ. In this case, by Lemma 2.2 (3), it computes a $(\frac{\log t}{\log(q/p^2)} \cdot \Delta/p)$-defective p^2-coloring in time

[1] Line 3 of Algorithm 4 invokes the algorithm of Linial. To speed up the computation of $O(\Delta^{3/2})$-coloring φ_i, the algorithm of Linial employs the $O(\Delta^2)$-coloring ϑ.

$(\frac{\log t}{\log(q/p^2)}) \cdot O(q)$. In particular, if φ' is a $\Delta^{O(1)}$-coloring, then Procedure Defective-Col-LEG employs φ' to compute a $(\frac{\log(\Delta^{O(1)})}{\log(q/p^2)} \cdot \Delta/p)$-defective p^2-coloring in time $(\frac{\log(\Delta^{O(1)})}{\log(q/p^2)}) \cdot O(q)$. In other words, if Procedure Defective-Col-LEG employs a $\Delta^{O(1)}$-coloring, then it computes an $O(\Delta^{3/4})$-defective $O(\Delta^{1/2})$-coloring in time $O(q) = O(\Delta^{\epsilon} \cdot p^2) = O(\Delta^{1/2+\epsilon})$.

The modified Procedure Mod-Trade-LEG proceeds as follows. It accepts as input a $\Delta^{O(1)}$-coloring ϑ. First, it computes an $O(\Delta^{3/4})$-defective $O(\Delta^{1/2})$-coloring ψ of the input graph in time $O(\Delta^{1/2+\epsilon})$ by Procedure Defective-Col-LEG as was described above. Next, the subgraphs $G_1, G_2, ...$ induced by color classes of ψ are legally colored with distinct palettes of size $O(\Delta^{3/4})$ each, using the KW iterative procedure in parallel on all subgraphs, in time $O(\Delta^{3/4} \cdot \log(\frac{\Delta^{O(1)}}{\Delta^{3/4}})) = O(\Delta^{3/4} \log \Delta)$. (Observe that the KW iterative procedure invoked on a subgraph G_i needs an initial coloring ϑ_i to start working. Here we restrict the coloring ϑ of the entire graph G to the vertex set V_i of the subgraph G_i. The resulting restricted coloring is called ϑ_i, and is employed by the KW iterative procedure as an initial coloring.) Then the colorings produced by the invocations of the KW iterative procedure are combined into a unified legal $O(\Delta^{5/4})$ coloring. The combining step requires no communication whatsoever. This completes the description of the procedure. Its pseudocode is provided below. It properties are given in the next lemma.

Algorithm 5. Procedure Mod-Trade-LEG (G , ϑ)

```
1: $\psi$ :=Defective-Col-LEG ($G, \vartheta, \Delta^{1/4}, \Delta^{3/4}$)   /* $\psi$ is an $O(\Delta^{3/4})$-defective $O(\Delta^{1/2})$-coloring */
2: for each subgraph $G_i$ induced by $\psi$, in parallel do
3:    $\varphi'_i$ := color $G_i$ with $O(\Delta^{3/4})$ colors using the KW iterative procedure
      /* use the restriction $\vartheta_i$ of the coloring $\vartheta$ to the vertex set $V_i$ of $G_i$ as initial coloring */
4: end for
5: for $i = 1, 2, ...$, in parallel do
6:    combine all colorings $\varphi'_i$ into a unified legal $O(\Delta^{5/4})$-coloring $\varphi$ of $G$
7: end for
8: return $\varphi$
```

Lemma 4.2. *Procedure Mod-Trade-LEG invoked with a $\Delta^{O(1)}$-coloring ϑ as input computes an $O(\Delta^{5/4})$-coloring φ in time $O(\Delta^{3/4} \log \Delta)$. Specifically, if ϑ is a $(c' \cdot \Delta^k)$-coloring, for constants $c' > 0$, $k \geq 2$, then φ employs at most $(\frac{\log(c' \cdot \Delta^k)}{\log \Delta^{\epsilon}} \cdot \Delta^{5/4}) = O(\frac{k}{\epsilon} \cdot \Delta^{5/4})$ colors.*

Suppose that Procedure Mod-Trade-LEG is invoked with a $(c^2 \cdot \Delta^{5/2})$-coloring as input, for a positive constant c to be determined later. Then it computes a $(\frac{\log(c^2 \cdot \Delta^{5/2})}{\log \Delta^{\epsilon}} \cdot \Delta^{5/4})$-coloring φ, for an arbitrary positive constant $\epsilon \leq 1/4$. For any constant c such that $c \geq \frac{2\log c + 5/2}{\epsilon}$ it holds that $\frac{\log(c^2 \cdot \Delta^{5/2})}{\log \Delta^{\epsilon}} \leq c$. (To satisfy the condition set $\epsilon = 1/8$, and c to be a sufficiently large constant.) Consequently, the resulting coloring φ is a $\lfloor c \cdot \Delta^{5/4} \rfloor$-coloring.

Next, we present a set-system free procedure, called Procedure Poly-Col, that computes an $O(\Delta^{5/4})$-coloring of the input graph *from scratch*, in time $\tilde{O}(\Delta^{3/4}) + \log^* n$. (This is in contrast to Procedure Defective-Col-LEG that accepts as input a legal $O(\Delta^2)$-coloring ϑ.) Procedure Poly-Col is very similar to Procedure Simple-Col (Algorithm 2.) The main difference is that in step 8 Procedure Poly-Col invokes the method Generic-Merge with Procedure Mod-Trade-LEG as an instantiation for Procedure Reduce instead of the KW Procedure. In addition, the variables $c_1, c_2, ..., c_\Delta$, and d are set to $\lfloor c \cdot \Delta^{5/4} \rfloor$ instead of $\Delta + 1$, where c is a constant as above. The pseudocode of the procedure is given below. Its properties are summarized in the next lemma.

Algorithm 6. Procedure Poly-Col(G)

```
1: $\{G_1, G_2, ..., G_\Delta\} := \Delta$-Forest-Decomposition($G$)    /* using PR Procedure */
2: for $i = 1, 2, ..., \Delta$ in parallel do
3:     $c_i := \lfloor c \cdot \Delta^{5/4} \rfloor$ ;  $\varphi_i :=$ 3-color($G_i$)  /* using GP Alg.;  each $G_i$ is a forest */
4: end for
5: $d := \lfloor c \cdot \Delta^{5/4} \rfloor$
6: Generic-Merge($P, G_1, G_2, ..., G_\Delta, \varphi_1, \varphi_2, ..., \varphi_\Delta, c_1, c_2, ..., c_\Delta$,Procedure Mod-Trade-
   LEG, $d$)
```

Lemma 4.3. *Procedure Poly-Col computes from scratch a legal coloring of G that employs at most $c \cdot \Delta^{5/4} = O(\Delta^{5/4})$ colors. Its running time is $O(\Delta^{3/4} \log^2 \Delta + \log^* n)$. Moreover, Procedure Poly-Col is set-system free.*

In what follows we devise a set-system free procedure for computing a $(\Delta + 1)$-coloring in $O(\Delta + \log^* n)$ time. In [6] the authors of the current paper devised a procedure, called *Procedure Delta-Color*, (henceforth, *Procedure Delta-Color-SL*), that accepts as input a graph G and a γ-coloring, $\gamma = O(\Delta^2)$, and computes a $(\Delta + 1)$-coloring of G in $O(\Delta)$ time. (See Lemma 2.2.) However, Procedure Delta-Col-SL employs set-systems. Next, we overview Procedure Delta-Col-SL from [6]. This procedure works as follows. If the number of colors in the coloring it accepts as input is $\gamma = O(\Delta)$, then a $(\Delta + 1)$-coloring of G is computed directly from the γ-coloring using the KW iterative procedure within $O(\Delta)$ time. Otherwise, the graph G is partitioned into vertex-disjoint subgraphs with maximum degrees $d < \Delta$, by invoking Procedure Defective-Col-LEG. Next, for each subgraph, an $O(d^2)$-coloring is computed using set-systems, by Linial's algorithm [18]. Then Procedure Delta-Col-SL is invoked recursively on each of the subgraphs. As a result we obtain a $(d + 1)$-coloring for each subgraph. (The recursion depth is $\lfloor \log^* \Delta \rfloor$. For $j = 1, 2, ..., \lfloor \log^* \Delta \rfloor$, in recursion level $\lfloor \log^* \Delta \rfloor - j$ it holds that $d = d_j = \Theta(\Delta / \Pi_{i=j}^{\log^* \Delta} (\log^{(i)} \Delta))$, i.e., $d_j = \Omega(\Delta^{1-\epsilon})$ for any constant $\epsilon > 0$.) These colorings are then merged into a unified legal $(\Delta + 1)$-coloring of the input graph. This completes the description of Procedure Delta-Col-SL. The running time of a recursion level j, $j = 1, 2, ..., \lfloor \log^* \Delta \rfloor$, is $O(d_j \cdot \log^{(j)} \Delta)$. Consequently, the overall running time is $O(\sum_{j=1}^{\log^* \Delta}(d_j \cdot \log^{(j)} \Delta)) = O(\Delta)$.

The only step in Procedure Delta-Col-SL that employs set-systems is the step that computes $O(d^2)$-colorings. Specifically, let $d_j = c \cdot \Delta / \Pi_{i=j}^{\log^* \Delta}(\log^{(i)} \Delta)$, for some fixed positive constant c. For $j = 1, 2, ..., \lfloor \log^* \Delta \rfloor$, Procedure Delta-Col-SL computes $O(d_j^2)$-colorings of subgraphs of degree $O(d_j)$. To this end it employs the algorithm of Linial [18]. We argue that if Procedure Delta-Col-SL accepts an $O(\Delta^{5/4})$-coloring instead of an $O(\Delta^2)$-coloring, then employing set systems is no longer required. Observe that as $d_j = \Omega(\Delta^{1-\epsilon})$, for any constant $\epsilon > 0$, it follows that for a sufficiently large Δ, $\Delta^{5/4} \le d_j^2$, for all $j = 1, 2, ..., \lfloor \log^* \Delta \rfloor$. Hence an $O(\Delta^{5/4})$-coloring is in particular an $O(d_j^2)$-coloring for all $j = 1, 2, ..., \lfloor \log^* \Delta \rfloor$. Therefore, invoking Procedure Delta-Col-SL with an $O(\Delta^{5/4})$-coloring as input instead of an $O(\Delta^2)$-coloring eliminates the need of computing $O(d_j^2)$-colorings of subgraphs. Indeed, for a subgraph H of degree d_j, $j \in \{1, 2, ..., \lfloor \log^* \Delta \rfloor\}$, the input coloring is already an $O(d_j^2)$-coloring of H.

To summarize, we obtained a variant of Procedure Delta-Col-SL, to which we will refer as Procedure Mod-Delta-LEG. This procedure accepts as input a graph G of maximum degree Δ, and an $O(\Delta^{5/4})$-coloring ϑ for G. The procedure returns a $(\Delta + 1)$-coloring, and it does so within $O(\Delta)$ time. (Observe that the running time of Procedure Mod-Delta-LEG is not greater than the running time of Procedure Delta-Col-SL when invoked with an $O(\Delta^{5/4})$-coloring as input. The two procedures perform exactly the same steps, except that Procedure Mod-Delta-LEG skips the invocation of the algorithm of Linial.)

To complete the algorithm it is only left to combine Procedure Poly-Col (that computes an $O(\Delta^{5/4})$-coloring from scratch) with Procedure Mod-Delta-LEG that reduces the number of colors to $\Delta + 1$. The resulting procedure will be referred to as Procedure Fast-Col. It accepts as input a graph G, and performs two steps. In the first step it computes an $O(\Delta^{5/4})$-coloring ϑ using Procedure Poly-Col. In the second step it invokes the set-system free variant (Procedure Mod-Delta-LEG) of Procedure Delta-Col-SL with the coloring ϑ as input. This procedure outputs a $(\Delta+1)$-coloring. The running time of Procedure Fast-Col is the sum of the running times of Procedure Poly-Col and Procedure Mod-Delta-LEG. The former is, by Lemma 4.3, $O(\Delta^{3/4} \log^2 \Delta) + \log^* n$, and the latter is $O(\Delta)$. Hence the overall running time of Procedure Fast-Col is $O(\Delta) + \log^* n$.

Theorem 4.4. *Procedure Fast-Col computes a $(\Delta + 1)$-coloring of the input graph G in $O(\Delta + \log^* n)$ time. Moreover, this computation is set-system free.*

Our results give rise to set-system free algorithms for a variety of problems. (Detailed proofs of these results will be provided in the full version of this paper.)

Corollary 4.5. *(1) For an arbitrarily small constant $\epsilon > 0$, and a parameter t, $1 < t \le \Delta^{1-\epsilon}$, one can compute an $O(\Delta \cdot t)$-coloring in $O((\Delta/t) + \log^* n)$ time.*
(2) A Maximal Independent Set can be computed in $O(\Delta + \log^* n)$ time.
(3) For the family of graphs with bounded arboricity $a(G) = o(\sqrt{\log n})$ a

Maximal Independent Set and (Δ+1)*-coloring can be computed in sublogarithmic time. Results* ***(1),(2)*** *and* ***(3)*** *are obtained using set-system free algorithms.*

References

1. Alon, N.: Eigen-values and expanders. Combinatorica 6(2), 83–96 (1986)
2. 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)
3. Arora, S., Safra, S.: Probabilistic Checking of Proofs: A New Characterization of NP. Journal of the ACM 45(1), 70–122 (1998)
4. Awerbuch, B., Goldberg, A.V., Luby, M., Plotkin, S.: Network decomposition and locality in distributed computation. In: Proc. of the 30th IEEE Annual Symposium on Foundations of Computer Science, pp. 364–369 (October 1989)
5. Barenboim, L., Elkin, M.: Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. In: Proc. of the 27th ACM Symp. on Principles of Distributed Computing, pp. 25–34 (2008)
6. Barenboim, L., Elkin, M.: Distributed ($\Delta + 1$)-coloring in linear (in Δ) time. In: Proc. of the 41th ACM Symp. on Theory of Computing, pp. 111–120 (2009), http://arXiv.org/abs/0812.1379v2 (2008)
7. Barenboim, L., Elkin, M.: Deterministic distributed vertex coloring in polylogarithmic time. In: Proc. of the 29th ACM Symp. on Principles of Distributed Computing, pp. 410–419 (2010)
8. Ben-Aroya, A., Ta-Shma, A.: A combinatorial construction of almost-Ramanujan graphs using the zig-zag product. In: Proc. of the 40th ACM Symp. on Theory of Computing, pp. 325–334 (2008)
9. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control 70(1), 32–53 (1986)
10. Dinur, I., Reingold, O.: Assignment Testers: Towards a Combinatorial Proof of the PCP Theorem. SIAM Journal on Computing 36(4), 975–1024 (2006)
11. Erdős, P., Frankl, P., Füredi, Z.: Families of finite sets in which no set is covered by the union of r others. Israel Journal of Mathematics 51, 79–89 (1985)
12. Feige, U., Goldwasser, S., Lovasz, L., Safra, S., Szegedy, M.: Interactive proofs and the hardness of approximating cliques. JACM 43(2), 268–292 (1996)
13. Goldberg, A., Plotkin, S.: Efficient parallel algorithms for ($\Delta + 1$)- coloring and maximal independent set problem. In: Proc. 19th ACM Symposium on Theory of Computing, pp. 315–324 (1987)
14. Goldberg, A., Plotkin, S., Shannon, G.: Parallel symmetry-breaking in sparse graphs. SIAM Journal on Discrete Mathematics 1(4), 434–446 (1988)
15. Kothapalli, K., Scheideler, C., Onus, M., Schindelhauer, C.: Distributed coloring in $\tilde{O}(\sqrt{\log n})$ bit rounds. In: 20th International Parallel and Distributed Processing Symposium (2006)
16. Kuhn, F.: Weak graph colorings: distributed algorithms and applications. In: Proc. 21st ACM Symp. on Parallel Algorithms and Architectures, pp. 138–144 (2009)
17. Kuhn, F., Wattenhofer, R.: On the complexity of distributed graph coloring. In: Proc. of the 25th ACM Symp. on Principles of Distributed Computing, pp. 7–15 (2006)
18. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)

19. Lubotzky, A., Philips, R., Sarnak, P.: Ramanujan graphs. Combinatorica 8, 261–277 (1988)
20. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing 15, 1036–1053 (1986)
21. Meir, O.: Combinatorial PCPs with Efficient Verifiers. In: Proc. of the 50th Annual IEEE Symp. on Foundations of Computer Science, pp. 463–471 (2009)
22. Panconesi, A., Rizzi, R.: Some simple distributed algorithms for sparse networks. Distributed Computing 14(2), 97–100 (2001)
23. Panconesi, A., Srinivasan, A.: On the complexity of distributed network decomposition. Journal of Algorithms 20(2), 581–592 (1995)
24. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM, Philadelphia (2000)
25. Reingold, O.: Undirected ST-connectivity in log-space. In: Proc. of the 37th ACM Symp. on Theory of Computing, pp. 376–385 (2005)
26. Reingold, O., Vadhan, S., Wigderson, A.: Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In: Proc. of the 41st Annual IEEE Symp. on Foundations of Computer Science, pp. 3–13 (2000)
27. Schneider, J., Wattenhofer, R.: A New Technique For Distributed Symmetry Breaking. In: Proc. of the 29th ACM Symp. on Principles of Distributed Computing, pp. 257–266 (2010)

Physical Expander in Virtual Tree Overlay*

Taisuke Izumi[1], Maria Gradinariu Potop-Butucaru[2], and Mathieu Valero[2]

[1] Graduate School of Engineering, Nagoya Institute of Technology, Nagoya, 466-8555, Japan
t-izumi@nitech.ac.jp

[2] Université Pierre et Marie Curie - Paris 6, LIP6 CNRS 7606, France
maria.gradinariu@lip6.fr, mathieu.valero@gmail.com

Abstract. In this paper, we propose a new distributed construction of constant-degree expanders motivated by their application in P2P overlay networks and in particular in the design of robust tree overlays. Our key result can be stated as follows. Consider a complete binary tree T and construct a random pairing Π between leaf nodes and internal nodes. We prove that the graph G_Π obtained from T by contracting all pairs (leaf-internal nodes) achieves a constant node expansion with high probability. In the context of P2P overlays our result can be interpreted as follows: if each physical node participating to the tree overlay manages a random pair that couples one virtual internal node and one virtual leaf node then the physical-node layer exhibits a constant expansion with high probability. We encompass the difficulty of obtaining the random tree virtualization by proposing a local, self-organizing and churn resilient uniformly-random pairing algorithm with $O(\log^2 n)$ running time. Our algorithm has the merit to not modify the original tree overlay (we just control the mapping between physical nodes and virtual nodes). Therefore, our scheme is general and can be easily extended to a large class of overlays.

1 Introduction

Background and Motivation. P2P networks are appealing for sharing/diffusing/searching resources among heterogeneous groups of users. Efficiently organizing users in order to achieve these goals is the main concern that motivated the study of overlay networks. In particular, *tree overlays* became recently attractive since efficient implementations of various P2P communication primitives such as content-based publish/subscribe or multicast are strongly tied to the hierarchical and acyclic properties of trees ([4, 7, 12, 20]). Therefore, many P2P variants of classical tree structures (e.g. B-trees, R-trees or P-trees) have been designed so far [1, 5, 6, 16, 17, 22]. However, the effectiveness of their use in real applications is shadowed by their vulnerability to faults and churn (i.e. frequent joins and leaves). Therefore, it is a challenging problem to lay the bases for designing robust tree overlays.

* This work is supported in part by KAKENHI no. 22700010 and Foundation for the Fusion of Science and Technology (FOST).

A promising way of measuring the robustness of overlay networks is the evaluation of the *graph expansion*. The (node) expansion $h(G)$ of an undirected graph $G = (V_G, E_G)$ is defined as:

$$h(G) = \min_{S \subseteq V_G \| S| \leq n/2} \frac{|\Gamma(S)|}{|S|},$$

where $\Gamma(S)$ is the set of nodes that are adjacent to a node in S but not contained in S. The implication of node expansion is that the deletion of at least $h(G) \cdot k$ nodes is necessary to disconnect a component of k nodes in G. That is, graphs with good expansion are hard to be partitioned into a number of large connected components. In this sense, the expansion of a graph can be seen as a good evaluation of its resilience to faults and churn. Interestingly, the expansion of tree overlays is trivially $O(1/n)$. This weakness to faults explains why tree overlays are not pervasive in real applications.

Our focus in this paper is to provide a mechanism to make tree overlays robust. In particular, we are interested in *generic* schemes applicable with minimal extra cost to a large class of tree overlays. As seen above, there is a broad class of tree-based data structures with different characteristics. However, their distributed implementations face similar difficulties when trying to circumvent the threat of disconnection. Therefore, providing a generic scheme for increasing their robustness would offer the substantial benefit for their systematic distributed implementation.

Our contribution. Solutions for featuring P2P tree overlays with robustness range from increasing the connectivity of the overlay (in order to eventually mask the network churn and faults) to adding an additional mechanism for monitoring and repairing the overlay. However the efficiency of these techniques is shadowed by the extra-cost needed to their implementation in dynamic settings. Moreover, the design of those mechanisms often depends on the specific tree overlay implementation, and thus their generalization is difficult. Therefore, we propose a totally novel approach that exploits the principle of *tree virtualization.* That is, in a tree overlay one physical node may be in charge of several virtual nodes. The core of our approach is to use this mapping between virtual and physical nodes such that the physical-node layer exhibits a good robustness property (e.g. constant expansion).

Our primary contribution is the following theorem which is the key in the construction of our random virtualization scheme:

Theorem 1. *Let T be a complete n-node binary tree with duplication of the root node (the duplicated root is seen to be identical to the original root). Then, we can define a bijective function Π from leaf nodes to internal nodes. Let G_Π be the graph obtained from T by contracting pairs $(v, \Pi(v))$ for all v* [1]*. If Π is chosen uniformly at random then G_Π has a constant (node) expansion with probability $1 - o(1)$.*

[1] The contraction of $(v, \Pi(v))$ means that we contract the edge $\{v, \Pi(v)\}$ assuming that it (virtually) exists in T.

An immediate consequence of this theorem is that the physical-node layer achieves a constant expansion with high probability if a random chosen couple composed of one leaf and one internal node is assigned to each physical node. It should be noted that our random tree virtualization does not modify the original properties of the tree overlay since we only control the mapping to physical nodes. This feature makes our scheme applicable to a large class of tree overlays.

Note that the above result heavily relies on the uniform random bijection (i.e. random perfect bipartite matching) between internal and leaf nodes in the tree overlay. Therefore, in order to prove the effectiveness of our proposal in a P2P context we also address the construction of random perfect bipartite matching over internal and leaf nodes. We propose a local and self-organizing scheme that builds upon the parallel random permutation algorithm of Czumaj et. al.[9]. Our scheme increases the graph expansion to a constant within $O(\log^2 n)$ synchronous rounds (n is the number of physical nodes). The quick convergence of our scheme in dynamic settings is validated through extended simulations.

Roadmap. In Section 2, we discuss the related work and focus mainly the distributed computing area. In Section 3 we present the proof of our main result. We discuss the distributed implementation of our scheme in Section 4 which also includes the simulation results using adversarial scenarios. Finally, in Section 5 we conclude and propose some future research directions.

2 Related Works

Expander graphs have been studied extensively in many areas of theoretical computer science. A good tutorial can be found in [15]. In the following we restrict our attention to distributed constructions with a special emphasis on specific P2P design.

There are several results related to expander construction in distributed settings. Most of those results are based on the distributed construction of random regular graphs, which exhibit a good expansion with high probability. To the best of our knowledge one of the first papers that addressed expander constructions in P2P settings is [18]. The authors compose d Hamiltonian cycles to obtain a $2d$-regular graph. In [21] the authors propose a fault-tolerant expander construction using a pre-constructed tree overlay. It provides the mechanism to maintain an approximate random d-regular graph under the assumption that the system always manages a spanning tree. The distributed construction of random regular graphs based on a stochastic graph transformation is also considered in [8, 13]. They prove that repeating a specific stochastic graph modification (e.g., swapping the two endpoints of a length-three path) eventually returns a uniformly-random sampling of regular graphs. Since all the previously mentioned algorithms are specialized in providing good expansion, the combination with the overlays maintenance is out of their scope. Therefore, these works cannot be easily extended to a generic fault tolerant mechanism in order to improve the resilience of a distributed overlay. Contrary to the previous mentioned works, our study can be seen as a way of identifying implicit expander properties in a given topological structure. There are several works along this direction. In [10] the authors propose a self-stabilizing constructions of spanders, which are spanning subgraphs including smaller number of edges than the original graph but having the asymptotically same expansion as the original (i.e. $O(\frac{1}{n})$)[2]. Recently, Pandurangan and Trehan propose a local and self-healing scheme to manage a good expansion for adversarial joins and leaves [19]. Abraham *et al.* [2] and Aspnes and Wieder [3] respectively give the analysis of the expansion for some specific distributed data structures (skip graphs and skip b-trees). Recently Goyal et.al. [14] prove that given a graph G, the composition of two random spanning trees has the expansion of at least $\Omega(h(G)/\log n)$, where n is the number of nodes in G. We can differentiate our result from the above works by its generality, the novelty of random tree virtualization concept and the constant expansion that is also maintained in the event of joins and leaves.

3 The Expander Property of G_Π

3.1 Notations

For an undirected graph G, V_G and E_G respectively denote the sets of all nodes and edges in G. Given a graph G and a subset of nodes $S \subseteq V_G$, we define Ind(S) to be the subgraph of G induced by S. For a set of nodes S, its complement is denoted by $\overline{S}$. The *node boundary* of a set $S \subseteq V_G$ is defined as a set of nodes in $\overline{S}$ that connect to at least one node in S, which is denoted by $\Gamma(S)$.

Let $T = (V_T, E_T)$ be a binary tree. The sets of leaf nodes and internal nodes for T are respectively denoted by $L(V_T)$ and $I(V_T)$. Given a subset $S \subseteq V_T$, we also define $L(S) = L(V_T) \cap S$ and $I(S) = I(V_T) \cap S$. For a (sub)tree X, the root node of X is denoted by $r(X)$, and the parent of $r(X)$ is denoted by $p(X)$.

3.2 Preliminary Results

In the following T denotes a complete binary tree without explicit statement. The root of T is denoted by $r(T)$. We prove several auxiliary results that will be further used in our main result.

Lemma 1. *For any nonempty subset $S \subseteq I(V_T)$ such that* Ind(S) *is connected, $|\Gamma(S)| \geq |S| + 1$. In particular, if $r(T) \notin S$ holds, $|\Gamma(S)| \geq |S| + 2$.*

The above lemma can be generalized for any (possibly disconnected) subset $S \subseteq I(V_T)$.

Lemma 2. *Given any nonempty subset $S \subseteq I(V_T) \setminus \{r(T)\}$ such that* Ind(S) *of T consists of m connected components, $|\Gamma(S)| \geq |S| + m + 1$ holds.*

The following corollary is simply deduced from Lemma 2.

Corollary 1. *Let X be a subtree of T. For any subset $S \subseteq I(V_X)$, $|\Gamma(S) \cap V_X| \geq |S \cap V_X|$. In particular, if S is nonempty, we have $|\Gamma(S) \cap V_X| \geq |S \cap V_X| + 1$.*

[2] A spander is also called a *sparsifier.*

3.3 Main Result

In what follows, $|L(V_T)|$ is denoted by n for short (i.e., n is the number of nodes in G_Π). We also assume Π is a bijective function from leaf nodes to the set of internal nodes (where the root doubly appears), which is chosen from all $n!$ possible functions uniformly at random. For a subset of nodes $S \subseteq L(V_T)$, we define $\Pi(S) = \{\Pi(u) | u \in S\}$ and $Q_\Pi = S \cup \Pi(S)$.

Provided a subset $S \subseteq L(V_T)$ satisfying $|S| < n/2$, we say a subtree X is S-*occupied* if all of its leaf nodes belong to S. An S-occupied subtree X is *maximal* if there is no S-occupied subtree X containing X as a subtree. Note that two S-occupied maximal subtrees X_1 and X_2 in a common tree T are mutually disjoint and $p(X_1) \neq p(X_2)$ holds because of their maximality. We first show two lemmas used in the main proof.

Lemma 3. *Let X be a maximal S-occupied subtree for a nonempty subset $S \subseteq L(V_T)$. Then, $|\Gamma(Q_\Pi) \cap V_X| \geq |\overline{Q_\Pi} \cap V_X|/2$ holds.*

Lemma 4. *Given a subset $S \subseteq L(V_T)$ such that $|S| \leq n/2$, let $X_0, X_1, \cdots X_k$ be all maximal S-occupied subtrees and $V_X = \cup_{i=1}^{k} V_{X_i}$. For any $\alpha < 1$,* $\Pr(|\Pi(S) \cap I(V_X)| \geq \alpha |I(V_X)|) \leq \binom{|S|}{\alpha(|S|-k)} \left(\frac{(|S|-k)}{n}\right)^{\alpha(|S|-k)}$.

The implication of the above two lemmas is stated as follows: We are focusing on a subset of boundary nodes $\Gamma(Q_\Pi)$ that are associated with some "hole" (that is, the set of nodes not contained in Q_Π) in S-occupied subtrees. Lemma 3 implies that at least half of the nodes organizing the hole belong to $\Gamma(Q_\Pi)$. Lemma 4 bounds the probability that S-occupied subtrees have the hole with size larger or equal to $(1-\alpha)|I(V_X)|$. We also use the following inequality:

Fact 1 (Jensen's inequality). *Let f be the convex function, $p_1, p_2, \cdots$ be a series of real values satisfying $\sum_{i=1}^{\infty} p_i = 1$, and $x_1, x_2, \cdots$ be a series of real values. Then, the following inequality holds:*

$$\sum_{i=1}^{\infty} p_i f(x_i) \geq f(\sum_{i=1}^{\infty} p_i x_i)$$

We give the proof of the main theorem (the statement is refined).

Theorem 1. *The node expansion of G_Π is at least $\frac{1}{480}$ with probability* $1 - o(1)$.

Proof. To prove the lemma, we show that with high probability, $|\Gamma(S)| \geq |S|/480$ holds for any subset $S \subseteq V_{G_\Pi}$ such that $|S| \leq n/2$. In the proof, we identify $L(T)$ and V_{G_Π} as the same set, and thus we often refer to S as a subset of $L(V_T)$ without explicit statement. Given a set S, let k be the number of maximal S-occupied subtrees, $\mathcal{X} = \{X_1, X_2, \cdots, X_k\}$ be all maximal S-occupied trees, and $V_X = \cup_{i=1}^{k} V_{X_i}$. We also define $P = \{p(X_i) | X_i \in \mathcal{X}\}$. Throughout this proof, we omit the subscript Π of Q_Π. For a subset $Y \subseteq V_T$, let $Q(Y) = Q \cap Y$ and $q(Y) = |Q(Y)|$ for short.

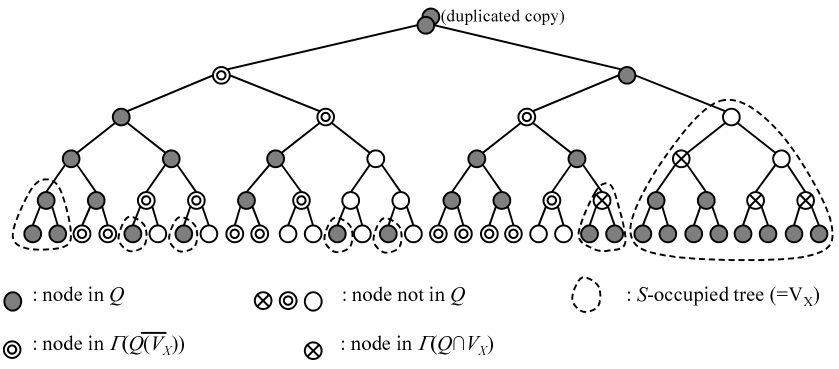

(duplicated copy)
: node in Q
: node not in Q
: S-occupied tree (=V_X)
: node in $\Gamma(Q(\overline{V_X}))$
: node in $\Gamma(Q \cap V_X)$

Fig. 1. Illustration of the set $\Gamma(S) \cap V_X$, $\Gamma(Q(\overline{V_X}))$, and their boundaries. in the proof of Theorem 1

The goal of this proof is to show that $|\Gamma(Q)| \geq |S|/240$ holds with high probability in T for any given S. It follows that $|\Gamma(S)| \geq |S|/480$ holds in G_Π. To bound $|\Gamma(Q)|$, we first consider the cardinality of two subsets $\Gamma(Q) \cap V_X$ and $\Gamma(Q \cap \overline{V_X})$ $(= \Gamma(Q(\overline{V_X})))$. See Fig. 1 for an example. While these subsets are not mutually disjoint, only the roots of S-occupied subtrees can be contained in both $\Gamma(Q) \cap V_X$ and $\Gamma(Q(\overline{V_X}))$. It implies

$$|(\Gamma(Q) \cap V_X) \cap \Gamma(Q(\overline{V_X}))| \leq q(P). \tag{1}$$

Lemma 2 and 3 lead to the following inequalities:

$$\begin{aligned}|\Gamma(Q) \cap V_X| &\geq \frac{1}{2}\left(\sum_{i=1}^{k} |\overline{Q} \cap V_{X_i}|\right) \\ &\geq (|S| - k - q(V_X))/2. \end{aligned} \tag{2}$$

$$\begin{aligned}|\Gamma(Q(\overline{V_X})| &\geq |Q(\overline{V_X})| + 1 \\ &\geq (|S| - 1) - q(V_X) + 1 = |S| - q(V_X). \end{aligned} \tag{3}$$

By inequalities 1, 2, and 3, we can bound the size of $\Gamma(Q)$ as follows:

$$\begin{aligned}|\Gamma(Q)| &\geq |(\Gamma(Q)) \cap V_X| + |\Gamma(Q(\overline{V_X}))| - q(P) \\ &\geq (|S| - k - q(V_X))/2 + |S| - q(V_X) - q(P) \qquad (4) \\ &\geq 3(|S| - k - q(V_X))/2 + k - q(P). \qquad (5)\end{aligned}$$

We consider the following two cases according to the value of k:

(Case 1). $k > |S|/16$: We show $|\Gamma(Q)| > |S|/240$ holds for any Π. If $q(P) \leq 12k/13$ holds, we have $|\Gamma(Q)| \geq k/13 \geq |S|/240$ from inequality 5 because of $q(V_X) \leq |S| - k$. Furthermore, if $|S| - q(V_X) - q(P) \geq |S|/240$, we have $|\Gamma(Q)| \geq |S|/240$ from inequality 4. Thus, in the following argument, we assume

$q(P) > 12k/13$ and $|S| - q(V_X) - q(P) < |S|/240$. Consider the subgraph H of T induced by $Q(P)$. We estimate the number of connected components in H to get a bound of $|\Gamma(V_H)|$ in T. Letting $\mathcal{C} = \{C_1, C_2, C_3, \cdots C_m\}$ be the set of connected components of H, we associate each leaf node u in a S-occupied subtree X_i with the component in $\mathcal{C}$ containing $p(X_i)$ (the node is associated with no component if $p(X_i)$ is not in $Q(P)$). Since each node $v \in H$ has one child belonging to V_X, each component in H forms a line graph monotonically going up to the root. Thus, if a component C_i has j nodes $u_1, u_2, \cdots u_j$, which are numbered from the leaf side, each node u_h $(1 \leq h \leq j)$ has a child as the root of a S-occupied tree having at least 2^{h-1} leaf nodes (recall that T is a complete binary tree). It follows that the number of nodes in S associated with C_i is greater or equal to $\sum_{h=1}^{j} 2^{h-1} = 2^j - 1$. Letting l_i be the number of components in $\mathcal{C}$ consisting of i nodes, we have:

$$\begin{aligned}|S|/m &\geq \sum_{i=0}^{n} \frac{l_i}{m}\left(2^i - 1\right)\\ &\geq 2^{\sum_{i=0}^{n} i\cdot(l_i/m)} - 1\\ &\geq 2^{\frac{12k}{13m}} - 1\\ &\geq 2^{\frac{3|S|}{52m}} - 1, \end{aligned} \tag{6}$$

where the second line is obtained by applying Jensen's inequality. To make the above inequality hold, the condition $|S|/m \leq 120 \Leftrightarrow m \geq |S|/120$ is necessary. Next, we calculate how many nodes in $\Gamma(V_H) \cap \overline{V_X}$ is occupied by Q. From the definition of H, $Q(\Gamma(V_H))$ does not contain any node in P (if a node $v \in P$ is contained, it will be a member of V_H and thus not in $\Gamma(V_H)$). Thus, any node in $\Gamma(V_H)$ is a member of V_X, $\overline{Q} \cap P$, or $\overline{V_X \cup P}$. Let $Y = Q(\overline{V_X \cup P} \cap \Gamma(V_H))$ and $y = |Y|$ for short. Since any node in $\overline{Q} \cap P$ is not contained in Q and the cardinality of $\Gamma(V_H) \cup V_X$ can be bounded by $q(P)$ (as the roots of S-occupied trees), we have the following bound from Lemma 2:

$$\begin{aligned}|\Gamma(Q)| &\geq |\Gamma(V_H) \setminus Q|\\ &\geq |\Gamma(V_H) \cap \overline{Q} \cap P|\\ &\geq |\Gamma(V_H)| - |\Gamma(V_H) \cap V_X| - |\Gamma(V_H) \cap (\overline{V_X \cup P})|\\ &\geq |\Gamma(V_H)| - q(P) - y\\ &\geq q(P) + m + 1 - q(P) - y\\ &\geq m - y.\end{aligned}$$

Since Y, $Q(P)$ and $Q(V_X)$ are mutually disjoint, we obtain $y + q(V_X) + q(P) \leq |S| \Leftrightarrow y \leq |S| - q(V_X) - q(P) < |S|/240$. Consequently, we obtain $|\Gamma(Q)| \geq |S|/240$.

(Case 2). $k \leq \frac{|S|}{16}$: In the following argument, given a set S satisfying $k \leq |S|/16$, we bound the probability $\Pr(|\Gamma(Q)| < |S|/32)$. From the inequality 5 and $k - q(P) \geq 0$, it follows $|\Gamma(Q)| \geq 3(15|S|/16 - q(V_X))/2$. In order to get $|\Gamma(Q)| \leq |S|/32$, we need $3(15|S|/16 - q(V_X))/2 \leq |S|/32 \Leftrightarrow q(V_X) \geq 11|S|/12$.

Thus, from Lemma 4, we can bound the probability as follows:

$$\begin{aligned}\Pr(|\mathit{\Gamma}(Q)| < |S|/32) &\leq \Pr(|\mathit{\Pi}(S)| \geq 11|I(V_X)|/12) \\ &\leq \binom{|S|}{11(|S|-k)/12}\left(\frac{|S|-k}{n}\right)^{11|S|/12} \\ &\leq \binom{|S|}{(|S|+k)/12}\left(\frac{|S|}{n}\right)^{11|S|/12}.\end{aligned}$$

Fixing k and $|S|$, we look at the number of possible choices of S. Since we can determine a S-occupied subtree X_i by choosing one node in T as its root, the set S is determined by choosing k nodes from all nodes in T. Thus, the total number of subsets S organizing at most $|S|/16$ S-occupied subtrees are bounded by $\sum_{i=1}^{|S|/16}\binom{2n}{i}$. Summing up this bound for any $|S| < n/2$, the total number is bounded by $\sum_{|S|=1}^{n/2}(32ne/|S|)^{|S|/16}$. Using the union bound, the well-known inequality $\binom{n}{m} \leq \sum_{i=1}^{m}\binom{n}{m} \leq (ne/m)^m$, and the condition $k < |S|/16$, we have:

$$\begin{aligned}&\Pr\left(\bigcup_{S\subseteq L(V_T)||S|\leq n/2} |\mathit{\Gamma}(Q)| < \frac{|S|}{32}\right) \\ &\leq \sum_{|S|=1}^{n/2}\left(\sum_{i=1}^{|S|/16}\binom{2n}{i}\right)\binom{|S|}{(|S|+k)/12}\left(\frac{|S|}{n}\right)^{11|S|/12}. \\ &\leq \sum_{|S|=1}^{n/2}\left(\frac{32ne}{|S|}\right)^{|S|/16}\left(\frac{192e}{17}\right)^{17|S|/192}\left(\frac{|S|}{n}\right)^{11|S|/12} \\ &\leq \sum_{|S|=1}^{n/2}\left((32e)^{1/16}(192e/17)^{17/192}\right)^{|S|}\left(\frac{|S|}{n}\right)^{(11/12-1/16)|S|}\end{aligned}$$

By numeric calculation, we have $\log((32e)^{1/16}(192e/17)^{17/192}) \leq 0.841$ and $(11/12 - 1/16) \geq 0.854$. Thus,

$$\leq \sum_{|S|=1}^{n/2} 2^{0.841|S|}\left(\frac{|S|}{n}\right)^{0.854|S|} = o(1).$$

The theorem is proved. □

A note on unbalnced trees. While the above theorem assumes that the tree T is balanced, that assumption is used only to lead the inequality 6. Considering an unbalanced tree where the difference of the depth among every pair of leaf nodes is at most c, we can guarantee that the number of nodes associated with a component C_i of size is j is greater or equal to $2^{j-c} - 1$. Thus, the inequality 6 is modified as $2^{\frac{3|S|}{52m}-c} - 1$. This provides a weaker constraint on the value of $|S|/m$, but showing a constant node expansion is still possible if c is constant.

4 Distributed Construction of G_Π

To prove the impact of Theorem 1 in P2P settings we have to construct a random bijection (i.e., random perfect bipartite matching) between internal and leaf nodes in tree overlays. In this section, we show that this distributed construction is possible with nice self-$*$ properties. That is, our scheme is totally-distributed, uses only local information, and is self-healing in the event of nodes joins and leaves. In the following we state the computational model and our network assumptions. Then, we propose our algorithm and study the impact of its performances under two churn scenario.

4.1 Computational Model

We consider a set of virtual nodes (peers) distributed over a connected physical network. Virtual nodes are structured in a binary tree overlay. Each physical node managing a virtual node v can communicate with any physical node managing v's neighbors in the overlay. The communication is synchronous and round-based. That is, the execution is divided into a sequence of consecutive rounds. All messages sent in some round are guaranteed to be received within the same round.

We assume that each physical node manages exactly one internal node and one leaf node in the virtual overlay. Moreover, we also assume that the tree is balanced. Note that these assumptions are not far from practice. Most of distributed tree overlay implementations embed balancing schemes. The preservation of matching structure is easily guaranteed by employing the strategy that one physical node always join as two new nodes. We can refer as an example the join/leave algorithm in [11], which is based on the above strategy and generally applicable to most of binary-tree overlays.

4.2 Uniformly-Random Matching Construction

The way leaf and internal nodes are matched via a physical node is generally dependent on the application requirements and is rarely chosen uniformly at random. That is, the implicit matching offered by the overlay may be extremely biased and the expansion factor computed in the previous section may not hold. Fortunately, the initial matching can be quickly "mixed" to obtain a uniformly-random matching. To this end we will extend the technique proposed by Czumaj et. al.[9] for fast random permutation construction to distributed scalable matchings in tree overlays. The following stochastic process rapidly mixes the sample space of all bipartite matchings between leafs and internal nodes:

1. Each leaf node first tosses a fair coin and decides whether it is active or passive.
2. Each active node randomly probes a leaf node and sends a matching-exchange request.

3. The passive node receiving exactly one matching-exchange request accepts it, and establishes the agreement to the sender of the request.
4. The internal nodes managed by the agreed pair are swapped.

Note that the above process is performed by all leaf nodes concurrently in a infinite loop. Following the analysis by Czumaj et. al.[9], the mixing time of the above process is $O(\log n)$.

The only point that may create problems in distributed P2P settings is the second step. Our solution to implement the random sampling mechanism with $O(\log n)$ time and message complexity is as follows: First, the prober sends a token to the root. From the root, the token goes down along the tree edges by selecting with equal probability one of its children. When the token reaches a leaf node, the destination is returned as the probe result.

Overall, the distributed scalable algorithm for constructing a random bipartite matching takes $O(\log^2 n)$ time. In the following subsection, we evaluate the performances of the above scheme face to churn.

4.3 Experimental Evaluation in Dynamic Environment

In this section, we experimentally validate the performance of our approach by simulation. In the simulation scenario the following four phases are repeated.

Nodes join. We assume that a newly-joining node knows the physical address of some leaf node u in the network. Let v and v' be the leaf and internal nodes that will be managed by the newly-joining physical node. The node u is replaced by a newly internal node v'. Then v and u become children of v'.

Nodes leave. The adversary chooses a number of nodes to make them leave. Since it is hard to simulate worst-case adversarial behavior, we adopt a heuristic strategy: given a physical node v with leaf v_L and internal node v_I, let $h(v)$ be the height of the smallest subtree containing both v_L and V_I. Intuitively, the physical node v with higher $h(v)$ has much contributition for avoiding the node boundary to be contained in a small subtree containing v_L. Following this intuition, the adversary always makes the node v with highest $h(v)$ leave.

Balancing. Most of tree-based overlay algorithms have some balancing mechanism. While the balancing mechanism has a number of variations, we simply assume a standard rotation mechanism. After a number of nodes join and leave, the tree is balanced by standard rotation operation.

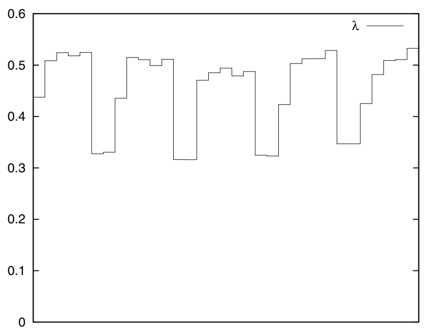
Matching Reconstruction. We run once the matching-mixing process described in the previous section.

Since exact computation of node expansion is coNP-complete, we monitor the second smallest eigenvalue λ of the graph's Laplacian matrix, which has a strong corelation to the node expansion: a graph with the second smallest eigenvalue λ is a $\lambda/2$-expander. In the following we propose our simulations results first in a churn free setting then in environments with different churn levels. Due to the space limitation the churn-free simulations are defered to the Appendix.

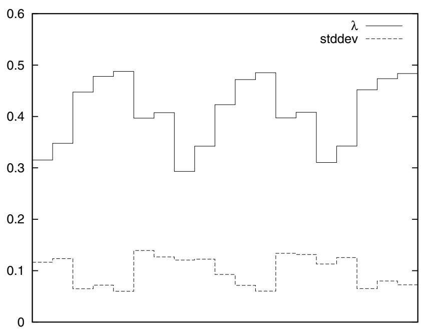
Churn free settings. We ran 100 simulations of 100 rounds with 512 nodes and no churn. The value of λ is calculated at the begin of each round. These simulations emphasize what is "expectable" from the mixing protocol and some of its dynamic properties.

λ varies from 0.263 to 0.524 with an average value of 0.502 and a standard deviation of 0.017. The low standard deviation and the closeness of average and maximum reached values of λ indicates that the minimum is rarely reached. Basically it is obtained when most nodes become responsible for internal nodes that are close to their leaves. Intuitively if each node is responsible for an ancestor of its leaf, there is no additionnal links between the left and the right subtrees of the root. In that case we do not take benefit of mixing and get bad expansion properties inherited from tree structures.

Churn prone settings. We ran 100 simulations of 100 rounds with 512 nodes and rates of churn fo 10% respectivelly 30%. Time is divided in seven rounds groups. During the first round a given percentage of new nodes join the system. During the second a given percentage of nodes leave the system. During the third round the tree is balanced and the mixing protocol is run. During other rounds the mixing protocol is run. λ is measured at the end of each round. Nodes gracefully leave the system. Those simulations investigate the impact of churn on λ and how fast our mixing protocol restores a stable configuration.



(a) λ over time with 10% of churn



(b) λ over time with 30% of churn

Figure 2a shows the evolution of λ over time in the presence of 10% of churn (10% is relative to the initial number of nodes). Each step stands for a round. From a stable configuration where λ oscillates between 0.52 and 0.48, it drops down to 0.4 every seven rounds due to arrivals and departures. The structure is sensitive to churn in the sense that it significantly decreases the value of λ. On the other hand, the proposed mixing protocol converges fast. It needs two rounds to reach the average expected value of λ . Note that the theoretical convergence was logarithmic in the number of nodes.

Figure 2b shows the evolution of λ over time in presence of 30% of churn (30% is relative to the initial number of nodes). Each step stands for a round.

Basically the increase of the churn stretches the curve. While the previous figure (2a) gives a good overview of the global behaviour of the protocol facing churn, the figure 2b emphases some interesting details. First, the mixing protocol is not monotonic; it might decrease λ. Second, the impact of arrivals and departures are distinct. Moreover, the magnitude of their impact is not predictable because the selection of bootstrap nodes is random. Starting from a stable situation arrivals will always decrease λ. With the proposed join mechanism, a new comer is weakly connected to the rest of the system. Starting from a stable situation graceful departures will almost always decrease λ. But with the proposed leave mechanism their impact is more subtle since they can largely modify the tree balance which also implies link exchanges. In some rare cases those exchanges (which could be thought as side effect shuffles of links) or the departure of weakly connected nodes could increase λ.

5 Concluding Remarks

We proposed for the first time in the context of overlay networks a generic scheme that transforms any tree overlay to an expander with constant node expansion with high probability. More precisely, we prove that a uniform random tree virtualization yields a node expansion of at least $\frac{1}{480}$ with probability $1 - o(1)$. Second, in order to demonstrate the effectiveness of our result in the context of P2P networks we further propose and evaluate in different churn scenario a simple scheme for uniform random tree virtualization in $O(\log^2 n)$ running time. Our scheme is totally distributed and uses only local information. Moreover, in the event of nodes join/leave or crash our scheme is self-healing.

The virtualization scheme itself is a promising approach and provides several interesting questions. We enumerate the open problems related to our result:

– **Better analysis of the matching convegence:** The simulations results show that the expansion computed is considerably larger than the theoretical bound. That is, the spectral expansion ranges from 1/4 to 1/2 while the theoretical bound is 1/480. In addition, the convegence time is also faster than the theoretical bound computed as logarithmic in the number of nodes. Finding tight bounds for both expansion and convegence time is an open problem.
– **Effective use of expander property:** In addition to fault resilience, expander graphs also offer the rapidly-mixing property of random walks. That is, MCMC-like sampling method effectively runs on our scheme. It is an interesting research direction to use the expansion property for implementing some statistical operations or query load balancing.
– **Application of virtualization scheme to other overlays:** The virtualization scheme can be easilly modified in order to be applied to other well-known overlays such as Chord or Pastry. Identifying the class of overlay networks where the virtualizaton scheme efficiently works is a challenging problem.

– **Self-healing strategies:** Another interesting research direction would be to exploit the constant expansion of the overlay in order to efficiently implement self-healing strategies.

References

1. Aberer, K., Cudre-Mauroux, P., Datta, A., Despotovic, Z., Hauswith, M., Punceva, M., Schmidt, R.: P-Grid: A self-organizing access structure for p2p information. In: Batini, C., Giunchiglia, F., Giorgini, P., Mecella, M. (eds.) CoopIS 2001. LNCS, vol. 2172, pp. 179–194. Springer, Heidelberg (2001)
2. Abraham, I., Aspnes, J., Yuan, J.: Skip B-trees. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 366–380. Springer, Heidelberg (2006)
3. Aspnes, J., Wieder, U.: The expansion and mixing time of skip graphs with applications. Distributed Computing 21(6), 385–393 (2008)
4. Baehni, S., Eugster, P.T., Guerraoui, R.: Data-aware multicast. In: DSN, pp. 233–242 (2004)
5. Bianchi, S., Felber, P., Potop-Butucaru, M.G.: Stabilizing distributed r-trees for peer-to-peer content routing. IEEE Trans. Parallel Distrib. Syst. 21(8), 1175–1187 (2010)
6. Caron, E., Desprez, F., Fourdrignier, C., Petit, F., Tedeschi, C.: A repair mechanism for fault-tolerance for tree-structured peer-to-peer systems. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2006. LNCS, vol. 4297, pp. 171–182. Springer, Heidelberg (2006)
7. Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A.I.T., Singh, A.: Splitstream: High-bandwidth content distribution in cooperative environments. In: SOSP, pp. 298–313 (2003)
8. Cooper, C., Dyer, M., Handley, A.: The flip markov chain and a randomizing p2p protocol. In: PODC, pp. 141–150 (2009)
9. Czumaj, A., Kutylowski, M.: Delayed path coupling and generating random permutations. Random Struct. Algorithms 17(3-4), 238–259 (2000)
10. Dolev, S., Tzachar, N.: Spanders: distributed spanning expanders. In: SAC, pp. 1309–1314 (2010)
11. du Mouza, C., Litwin, W., Rigaux, P.: SD-Rtree: A scalable distributed rtree. In: ICDE, pp. 296–305 (2007)
12. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., Kermarrec, A.-M.: Lightweight probabilistic broadcast. ACM Trans. Comput. Syst. 21(4) (2003)
13. Feder, T., Guetz, A., Mihail, M., Saberi, A.: A local switch Markov chain on given degree graphs with application in connectivity of peer-to-peer networks. In: FOCS, pp. 69–76 (2006)
14. Goyal, N., Rademacher, L., Vempala, S.: Expanders via random spanning trees. In: SODA, pp. 576–585 (2009)
15. Hoory, S., Linial, N., Wigderson, A.: Expendar graphs and their applications. Bull. Amer. Math. Soc. (43), 439–561 (2006)
16. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: a balanced tree structure for peer-to-peer networks. In: VLDB, pp. 661–671 (2005)
17. Jagadish, H.V., Ooi, B.C., Vu, Q.H., Zhang, R., Zhou, A.: Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In: ICDE, p. 34 (2006)

18. Law, C., Siu, K.-Y.: Distributed construction of random expander networks. In: IEEE Infocom, pp. 2133–2143 (2003)
19. Pandurangan, G., Trehan, A.: Xheal: Localized Self-healing using Expanders. In: PODC, pp. 301–310 (2011)
20. Paris, C., Kalogeraki, V.: A topologically-aware overlay tree for efficient and low-latency media streaming. In: QShine/AAA-IDEA. LNICST, vol. 22, pp. 383–399 (2009)
21. Reiter, M.K., Samar, A., Wang, C.: Distributed construction of a fault-tolerant network from a tree. In: SRDS, pp. 155–165 (2005)
22. Zhang, C., Krishnamurthy, A., Wang, R.Y.: Brushwood: Distributed trees in peer-to-peer systems. In: ITPTPS, pp. 47–57 (2005)

A Omitted Proofs

A.1 Proof of Lemma 1

Proof. We prove the case of $r(T) \notin S$ by induction on the cardinality of S. (**Basis**) If $|S| = 1$, the lemma clearly holds because every internal node except for the root of T has degree three. (**Inductive step**) Suppose as the induction hypothesis that $|\Gamma(S)| \geq |S| + 2$ holds for any connected set S of size k. we prove that for any $v \in I(V_T)$ that is connected to a node in S, $|\Gamma((S \cup \{v\}))| \geq |S \cup \{v\}| + 2$ holds. For short, let $S' = S \cup \{v\}$. Since S is connected, v is connected to exactly one node in S. In other words, node v has two neighbors of v in neither S nor $\Gamma(S)$, which are elements of $\Gamma(S')$. In contrast, v is an element of $\Gamma(S)$ but not in $\Gamma(S')$. It follows that $\Gamma(S') \geq \Gamma(S) - 1 + 2$ holds. From the induction hypothesis, we obtain $\Gamma(S') \geq \Gamma(S) + 1 \geq |S| + 1 + 2 = |S'| + 2$. The case $r(T) \in S$ is obviously deduced from the case of $r(T) \notin S$. The lemma is proved. □

A.2 Proof of Lemma 2

Proof. Let $C_1, C_2, \cdots C_j, \cdots C_m$ be the set of connected components in $\mathrm{Ind}(S)$. In the case of $r(T) \in S$, we assume $r(T) \in C_1$ without loss of generality. We prove the lemma by induction on j. (**Basis**) It holds from Lemma 1 (**Inductive step**) Suppose $|\Gamma(S)| \geq |S| + j + 1$ as the induction hypothesis. Consider adding a new component C_{j+1} into S. Let c be the number of nodes in C_{j+1}. Since $r(T) \notin C_{j+1}$, from Lemma 1, $|\Gamma(V_{C_{m+1}})| \geq c + 2$ holds. At most one node is shared by $\Gamma(S)$ and $\Gamma(V_{C_{m+1}})$, we have $|\Gamma(S \cup V_{C_{j+1}})| \geq |S| + j + 1 + c + 2 - 1 \geq (|S| + c) + (j + 1)$. The lemma is proved. □

A.3 Proof of Lemma 3

Proof. We omit the subscript Π of Q_Π for short. We divides $\overline{Q} \cap V_X$ into three mutually-disjoint subset $\overline{S_1}$, $\overline{S_2}$, and $\overline{S_3}$: Let $\overline{S_1} \subseteq V_X \cap \overline{Q}$ be the set of nodes that have no neighbor belonging to $Q \cap V_X$, $\overline{S_2} = \Gamma(\overline{S_1}) \cap \overline{Q}$, and $\overline{S_3} = (\overline{Q} \cap V_X) \setminus (\overline{S_1} \cup \overline{S_2})$ (see Fig. 2). Since X is S-occupied, $\overline{S_2}$ consists only of internal nodes. Thus, from Corollary 1, $|\overline{S_2}| = |\Gamma(\overline{S_1})| \geq |\overline{S_1}|$ holds. By the definition of $\overline{S_2}$ and $\overline{S_3}$, $\overline{S_2} \subseteq (\Gamma(Q)) \cap V_X$ and $\overline{S_3} \subseteq (\Gamma(Q)) \cap V_X$ hold. Consequently, we have $2|\Gamma(Q) \cap V_X| \geq 2(|\overline{S_2}| + |\overline{S_3}|) \geq |\overline{S_2}| + 2|\overline{S_3}| + |\overline{S_1}| \geq |\overline{Q} \cap V_X|$. The lemma is proved. □

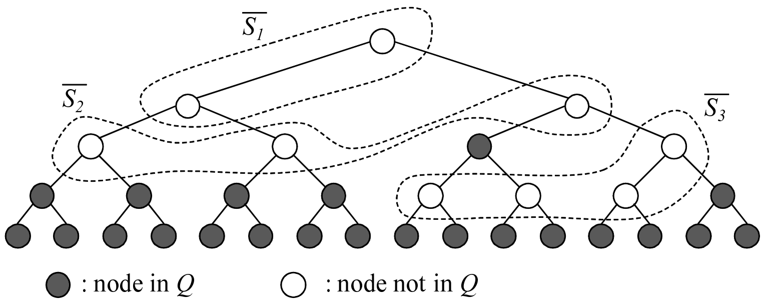


Fig. 2. Illustration of $\overline{S_1}$, $\overline{S_2}$, and $\overline{S_3}$ in the proof of Lemma 3

A.4 Proof of Lemma 4

Proof. Let $Z = I(V_X)$ for short. First, we fix a subset $S' \subseteq$ of S with cardinality $\alpha|Z|$, and consider the probability that $\Pi(S') \subseteq Z$ holds. Without loss of generality, we can regard Π as a permutation on Z whose first $|S|$ elements are mapped to Z. Thus, to compute the probability, it is sufficient to count the number of permutations where any element in S' appears at the first $|Z|$ elements of Π. Dividing the counted number by $n!$, the probability can be calculated as follows:

$$\begin{aligned}\Pr(S' \subseteq X) &= \frac{1}{n!}\binom{n-|S'|}{|Z|-|S'|}|Z|!(n-|Z|)! \\ &= \frac{|Z|(|Z|-1)(|Z|-2)\cdots(|Z|-\alpha|Z|+1)}{n(n-1)(n-2)\cdots(n-\alpha|Z|+1)} \leq \left(\frac{|Z|}{n}\right)^{\alpha|Z|}.\end{aligned}$$

Using the union bound, we can obtain the following bound:

$$\Pr(\Pi(S) \geq \alpha|Z|) \leq \Pr\left(\bigcup_{S' \subset S | S' = \alpha|Z|} S' \subseteq Z\right) \leq \binom{|S|}{\alpha|Z|}\left(\frac{|Z|}{n}\right)^{\alpha|Z|}.$$

Since $|Z| = |S| - k$ holds, the lemma is proved. □

Sub-logarithmic Test-and-Set against a Weak Adversary*

Dan Alistarh[1] and James Aspnes[2]

[1] EPFL
[2] Yale University

Abstract. A randomized implementation is given of a test-and-set register with $O(\log\log n)$ individual step complexity and $O(n)$ total step complexity against an oblivious adversary. The implementation is linearizable and multi-shot, and shows an exponential complexity improvement over previous solutions designed to work against a strong adversary.

1 Introduction

A *test-and-set* object supports an atomic test-and-set operation, which returns 0 to the first process that executes it and 1 to all subsequent processes. Test-and-set is a classic synchronization primitive, often used in multiprocessing computer systems as a tool for implementing mutual exclusion. It also has close connections to the traditional distributed computing problems of *consensus* [11] and *renaming* [2]. For two processes, test-and-set can be used to solve consensus and vice versa [11]; this implies that test-and-set has no deterministic wait-free implementation from atomic registers. Nonetheless, randomized implementations can solve test-and-set efficiently.

The randomized test-and-set object of Afek *et al.* [1] requires $O(\log n)$ steps on average, where n is the number of processes. It is built from a tree of 2-process test-and-set objects that are in turn built from 2-process randomized consensus protocols. The performance bounds hold even when scheduling is under the control of an *adaptive adversary*, which chooses at each step which process executes the next low-level operation based on complete knowledge of the system, including the internal states of processes.

In the case of consensus, it is known that replacing an adaptive adversary with an *oblivious adversary*, that fixes the entire schedule in advance, improves performance exponentially, from an $\Omega(n)$ lower bound on the expected number of steps performed by any one process [3] to an $O(\log n)$ upper bound [6]. Thus, a natural question is whether an algorithm with step complexity lower than $\Theta(\log n)$ is possible for test-and-set against a weak adversary.

In this paper, we answer this question in the affirmative. We show that, even though test-and-set has a fast $O(\log n)$ implementation against an adaptive adversary, this same exponential improvement holds: by exploiting the limitations of the oblivious adversary, we can reduce the complexity of test-and-set from $O(\log n)$ to $O(\log\log n)$.

* The work of Dan Alistarh was supported by the NCCR MICS Project. The work of James Aspnes was supported in part by NSF grant CCF-0916389.

The essential idea of our algorithm is to rapidly eliminate most processes from consideration using a sequence of *sifting* rounds, each of which reduces the number of survivors to roughly the square root of the number of processes that enter the round, with high probability; in particular, this reduces the number of survivors to polylogarithmic in $O(\log\log n)$ rounds.

The intuition behind the sifting technique is quite simple: each process either writes or reads a register in each round with a carefully-tuned probability. The process continues to the next round only if it chooses to write, or if it reads the value $\perp$, indicating that its read preceded any writes in that round. Because an oblivious adversary cannot predict which process will read and which will write, it effectively plays a game where the processes access the register one at a time, with only writers surviving after the first write; the probabilities are chosen so that the sum of the expected number of initial readers and the expected number of writers is minimized. At the same time, this scheme ensures that *at least one* process survives each round of sifting, because either all writers survive, or, if there are no writers, all readers survive. This technique works despite asynchrony or process crashes.

After $\Theta(\log\log n)$ rounds of sifting, the number of remaining candidates is small enough that the high-probability bounds used to limit the number of survivors in each round stop working. On the other hand, we notice that in this case we are left with $O(\text{polylog}\, n)$ survivors, and we can feed these survivors into a second stage consisting of the adaptive test-and-set implementation of [2], that has step complexity $O(\log k)$ for k participating processes. Thus, the running time of this second stage is also $O(\log\log n)$, which yields the step complexity upper bound of $O(\log\log n)$. A similar analysis shows that the total number of steps that all processes take during an execution of the algorithm is $O(n)$, which is clearly optimal.

It is worth noting that in the presence of an adaptive adversary, though the initial sifting phase fails badly (the adversary orders all readers before all writers, so all processes survive each round), the adaptive test-and-set still runs in $O(\log n)$ time, and the $O(\log\log n)$ overhead of the initial stage disappears into the constant. So our algorithm has the desirable property of degrading gracefully even when our assumption of an oblivious adversary is too strong.

While it is difficult to imagine an algorithm with significantly less than $O(\log\log n)$ step complexity, the question of whether a better algorithm is possible remains open. This is also the case for the adaptive adversary model, where there is no lower bound on expected step complexity to complement the $O(\log n)$ upper bounds of [1, 2].

The step complexity of our algorithm is $O(\log\log n)$ in expectation, and $O(\log n)$, with high probability. We notice that, even against a weak adversary, any step complexity upper bound on test-and-set that holds with high probability has to be at least $\Omega(\log n)$. This result follows from a lower bound of Attiya and Censor-Hillel on the complexity of randomized consensus [4].

Also of note, our algorithm suggests that non-determinism can be used to avoid some of the cost of expensive synchronization in shared memory, if the scheduler is oblivious. More precisely, Attiya *et al.* [5] recently showed that deterministic implementations of many shared objects, including test-and-set, queues, or sets, have worst-case executions which incur expensive read-after-write (RAW) or atomic-write-after-read (AWAR) operation patterns. In particular, ensuring RAW order in shared memory requires introducing memory fences or barriers, which are considerably slower than regular instructions.

First, we notice that their technique also applies to randomized read-write algorithms against an adaptive adversary, yielding an adversarial strategy that forces *each* process to perform an expensive RAW operation pattern with probability 1. On the other hand, the sifting procedure of our algorithm bounds the number of processes that may perform RAW patterns in an execution to $O(\sqrt{n})$, with high probability. This shows that randomized algorithms can avoid part of the synchronization cost implied by the lower bound of [5], as long as the scheduler is oblivious.

Roadmap. We review the related work in Section 2, and precisely define the model of computation, problem statement, and complexity measures in Section 3. We then present our algorithm and prove it correct in Section 4. In Section 4.4, we present a simple technique for turning the single-shot test-and-set implementation into a multi-shot one, and derive lower bounds in Section 5. We summarize our results and give an overview of directions for future work in Section 6.

2 Related Work

The test-and-set instruction has been present in hardware for several decades, as a simple means of implementing mutual exclusion. Herlihy [11] showed that this object has consensus number 2.

Several references studied wait-free randomized implementations of test-and-set. References [10, 14] presented implementations with super-linear step complexity. (Randomized consensus algorithms also implement test-and-set, however their step complexity is at least linear [3].) The first randomized implementation with logarithmic step complexity was by Afek *et al.* [1], who extended the tournament tree idea of Peterson and Fischer [13], where the tree nodes are two-process test-and-set (consensus) implementations as presented by Tromp and Vitanyi [15]. Their construction has expected step complexity $O(\log n)$. This technique was made adaptive by the RatRace protocol of [2], whose step complexity is $O(\log^2 k)$ with probability $1 - 1/k^c$, for c constant, where k is the actual number of processes that participate in the execution. We use the RatRace protocol as the final part of our test-and-set construction. Note that these previous constructions assume a strong adaptive adversary. The approaches listed above for sublinear randomized test-and-set incur cost at least logarithmic in terms of expected time complexity, even if the adversary is oblivious, since they build on the tournament tree technique. References [7, 8] give deterministic test-and-set and compare-and-swap implementations with constant complexity in terms of *remote memory references* (RMRs), in an asynchronous shared-memory model with no process failures (by contrast, our implementation is wait-free). The general strategy behind their test-and-set implementation is similar to that of this paper and that of Afek *et al.* [1]: the algorithm runs a procedure to elect a leader process, i.e. the winner of the test-and-set, and then uses a separate flag register to ensure linearizability.

3 Preliminaries

Model. We assume an asynchronous shared memory model in which at most n processes may participate in any execution, $t < n$ of which may fail by crashing. We assume that processes know n (or a rough upper bound on n). Processes communicate through multiple-writer-multiple-reader atomic registers. Our algorithms are randomized, in that the processes' steps may depend on random local coin flips. Process crashes and scheduling are controlled by a weak *oblivious* adversary, i.e. an adversary that cannot observe the results of the random coin flips of the processes, and hence has to fix its schedule and failure pattern before the execution. On the other hand, we assume that the adversary knows the structure of the algorithm. By contrast, a strong *adaptive* adversary (as considered in Lemma 5) knows the results of the coin flips by the processes at any point during the algorithm, and may adjust the scheduling and failure pattern accordingly.

Problem Statement. The multi-use *test-and-set* bit has a test-and-set operation which atomically reads the bit and sets its value to 1, and a reset operation which sets the bit back to 0. We say that a process *wins* a test-and-set object if it reads 0 from the object; otherwise, if it reads 1, the process *loses* the test-and-set. By the sequential specification, each correct process eventually returns an indication (*termination*), and only one process may return winner from a single instance of test-and-set (the *unique winner* property). Also, no process may return loser before the winner started the execution, and only the winner of an instance may successfully reset the instance.

Complexity Measures. We measure complexity in terms of process steps: each shared-memory operation and (local) coin flip is counted as a step. The *total step complexity* counts the total number of process steps in an execution, while the *individual step complexity* (or simply *step complexity*) is the number of steps a single process may perform during an execution.

As a second measure, we also consider the number of read-after-write (RAW) patterns that our algorithm incurs. This metric has been recently analyzed by Attiya et al. [5] in conjunction with atomic write-after-read (AWAR) operations (we consider read-write algorithms, which cannot employ AWAR patterns). In brief, the RAW pattern consists of a process writing to a shared variable A, followed by the same process reading from a different shared variable B, without writing to B in between. Enforcing RAW order on modern architectures requires introducing memory fences or barriers, which are substantially slower than regular instructions. For a complete description of RAW/AWAR patterns, please see [5].

4 Test-and-Set Algorithm

In this section, we present and prove correct a randomized test-and-set algorithm, called Sift, with expected (individual) step complexity $O(\log \log n)$ and total step complexity $O(n)$. The algorithm is structured in two phases: a sifting phase, which eliminates a large fraction of the processes from contention, and a competition phase, in which the survivors compete in an adaptive test-and-set instance to assign a winner.

```
1  Shared:
2  $Reg$, a vector of atomic registers, of size $n$, initially $\perp$
3  $Resolved$, an atomic register

4  procedure Test-and-Set()
5     if $Resolved$ = true then
6         return loser
      /* sifting phase */
7     for round $r$ from $0$ to $\lceil \frac{5}{2} \ln \ln n \rceil$ do
8         $\pi_r \leftarrow n^{-(2/3)^r/2}$
9         $flip \leftarrow 1$ with probability $\pi_r$, $0$ otherwise
10        if $flip = 1$ then
11            $Reg[r] \leftarrow p_i$
12        else
13            $val \leftarrow Reg[r]$
14            if $val \neq \perp$ then
15                $Resolved \leftarrow$ true
16                return loser
      /* competition phase */
17    $result \leftarrow$ RatRace($p_i$)
18    return $result$
```

Fig. 1. The Sift test-and-set algorithm

4.1 Description

The pseudocode of the algorithm is presented in Figure 1. Processes share a vector Reg of atomic registers, initially $\perp$, and an atomic register $Resolved$, initially false. The algorithm proceeds in two phases.

The first, called the *sifting* phase, is a succession of rounds $r \geq 0$, with the property that in each round a fraction of the processes are eliminated. More precisely, in round r, each process flips a binary biased coin which is 1 with probability $\pi_r = n^{-(2/3)^r/2}$ and 0 otherwise (line 9). If the coin is 1, then the process writes its identifier p_i to the register $Reg[r]$ corresponding to this round, which is initially $\perp$. Every process that flipped 0 then reads the value of $Reg[r]$ in round r. If this value is $\perp$, then the process continues to the next round. Otherwise, the process returns loser, but first marks the *Resolved* bit to true, to ensure that no incoming process may win after a process has lost (lines 13-16). We will prove that by the end of the $\lceil \frac{5}{2} \ln \ln n \rceil + 1$ rounds in this phase, the number of processes that have not yet returned loser is $O(\log^7 n)$, with high probability.

In the *competition* phase, we run an instance of the RatRace [2] adaptive test-and-set protocol, to determine the one winner among the remaining processes. In brief, in RatRace, each process will first acquire a temporary name, and then compete in a series of two-process test-and-set instances to decide the winner. Since the number of processes that participate in this last phase is polylogarithmic, we will obtain that the number of steps a process takes in this phase is $O(\log \log n)$ in expectation.

4.2 Proof of Correctness

We first show that the algorithm is a correct test-and-set implementation.

Lemma 1 (Correctness). *The* Sift *algorithm is a linearizable test-and-set implementation.*

Proof. The *termination* property follows by the structure of the sifting phase, and by the correctness of the RatRace protocol [2]. The *unique winner* property also follows from the properties of RatRace. Also, notice that, by the structure of the protocol, any process that performs alone in an execution returns winner.

To prove linearizability, we first show that the algorithm successufully elects a leader, i.e., given an execution $\mathcal{E}$ of the protocol and a process p_ℓ that returns loser in $\mathcal{E}$, there exists a (unique) process $p_w \neq p_\ell$ such that p_w either returns winner in $\mathcal{E}$, or crashes in $\mathcal{E}$. Second, we show that, using the $Resolved$ bit, the test-and-set operation by process p_w can be linearized *before* p_ℓ's test-and-set operation.

For the first part, we start by considering the line in the algorithm where process p_ℓ returned loser. If p_ℓ returned on line 18, then the above claim follows by the linearizability of the RatRace test-and-set implementation [2]. On the other hand, if p_ℓ returned on line 16, it follows that p_ℓ has read the identifier of another process from a shared register $Reg[r]$ in some round $r \geq 1$. Denote this process by q_1. We now analyze the return value of process q_1 in execution $\mathcal{E}$. If q_1 crashed or returned winner in execution $\mathcal{E}$, then the claim holds, since we can linearize q_1's operation or crash before p_ℓ's operation. If q_1 returns loser from RatRace, then again we are done, by the linearizability of RatRace. Therefore, we are left with the case where q_1 returned loser on line 16, after reading the identifier of another process q_2 from a register $Reg[r']$ with $r' \geq r$. Notice that q_2 and p_ℓ are distinct, since the write operations are atomic.

Next, we can proceed inductively to obtain a maximal chain of *distinct* processes q_1, q_2, up to q_k for some $k \geq 1$ such that for any $1 \leq i \leq k-1$, process q_i read process q_{i+1}'s identifier and returned loser in line 16. This chain is of length at most $n-1$. Considering the last process q_k, since the chain is maximal, process q_k could not have read any other process's identifier in line 13 during the sifting phase. Therefore, process q_k either obtains a decision value from RatRace in line 18, or crashes in $\mathcal{E}$ after reading $Resolved =$ false in line 6 of the protocol. Notice that, since the $Resolved$ bit is atomic, q_k's test-and-set operation could not have started after p_ℓ's operation ended. Therefore, if q_k decides winner or crashes, then we can linearize its operation before p_ℓ's operation and we are done. Otherwise, if q_k decides loser from RatRace, then there exists another process p_w that either returns winner or crashes during RatRace, and whose RatRace(p_w) operation can be linearized before q_k's. Therefore, we can linearize p_w's test-and-set operation before p_ℓ's to finish the proof of this claim.

Based on this claim, we can linearize any execution in which some process returns loser as follows. We consider the losing processes in the order of their read operations on register $Resolved$. Let p_ℓ be the first losing process in this order. We then apply the claim above to obtain that there exists a process p_w that either crashes or returns winner, whose test-and-set operation can be linearized before that of p_ℓ. This defines a valid linearization order on the operations that return in execution $\mathcal{E}$. The other operations (by processes that crash, except p_w) may be linearized in the order or non-overlapping operations. The remaining executions can be linearized trivially. We note that, since we use the *Resolved* bit to ensure linearization, we avoid the linearizability issues recently pointed out by Golab et al. [9] for randomized implementations.

4.3 Proof of Performance

We now show that the algorithm has expected step complexity $O(\log \log n)$. The intuition behind the proof is that, for each round r, the number of processes that keep taking steps *after* round $r+1$ of the sifting phase is roughly $\sqrt{n_r}$ (in expectation), where n_r is the number of processes that take steps in round $r+1$. Iterating this for $\lceil (5/2) \ln \ln n \rceil$ rounds leaves at most $\operatorname{polylog} n$ active processes at the end of the sifting phase, with high probability. We begin the proof by showing that the sifting phase reduces the set of competitors as claimed.

Lemma 2. *With probability* $1 - o(n^{-c})$*, for any fixed* c*, at most* $\ln^7 n$ *processes leave the sifting phase without returning* loser.

Proof. Fix some c, and let $c' = c + 1$.

Let

$$\begin{aligned}
\kappa &= -\frac{1}{\ln(2/3)}\left(1 - \frac{\ln 7 + \ln\ln\ln n}{\ln\ln n}\right). \\
&\le -\frac{1}{\ln(2/3)} \\
&< \frac{5}{2}.
\end{aligned}$$

We will show that, with high probability, it holds that for all $0 \le r \le \kappa \ln \ln n$, at most $n_r = n^{(2/3)^r}$ processes continue after r rounds of the sifting phase. The value of κ is chosen so that

$$\begin{aligned}
n_{\kappa \ln\ln n} &= n^{(2/3)^{\kappa \ln\ln n}} \\
&= \exp\left(\ln n \cdot (2/3)^{\kappa \ln\ln n}\right) \\
&= \exp\left(\exp\left(\ln\ln n + \ln(2/3) \cdot \kappa \ln\ln n\right)\right) \\
&= \exp\left(\exp\left(\ln\ln n - \left(1 - \frac{\ln 7 + \ln\ln\ln n}{\ln\ln n}\right)\ln\ln n\right)\right) \\
&= \exp\left(\exp\left(\ln\ln n - \ln\ln n + \ln 7 + \ln\ln\ln n\right)\right) \\
&= \exp\left(7 \ln\ln n\right) \\
&= \ln^7 n.
\end{aligned}$$

This bound is a compromise between wanting the number of processes leaving the sifting phase to be as small as possible, and needing the number of survivors at each stage to be large enough that we can characterize what happens to them using standard concentration bounds. Because $\kappa < \frac{5}{2}$, and extra rounds of sifting cannot increase the number of surviving processes, if there are at most $\ln^7 n$ survivors after $\kappa \ln\ln n + 1$

rounds, there will not be any more than this number of survivors when the sifting phase ends after $\left\lceil \frac{5}{2} \ln \ln n \right\rceil + 1$ rounds, establishing the Lemma.

We now turn to the proof of the n_r bound, which proceeds by induction on r: we prove that if fewer than n_r processes enter round $r+1$, it is likely that at most $n_{r+1} = n^{(2/3)^{r+1}}$ processes leave it. The base case is that at most $n_0 = n^{(2/3)^0} = n$ processes enter round 1.

Note that π_r, the probability of writing the register in round r, is chosen so that $\pi_r = n_r^{-1/2}$.

From examination of the code, there are two ways for a process to continue the execution after round $r+1$: by writing the register (with probability $\pi_r = n^{-(2/3)^r/2}$), or by reading $\perp$ from the register before any other process writes it. Suppose at most n_r processes enter round r. Then the expected number of writers is at most $n_r \pi_r = n_r^{1/2}$. On the other hand, since the adversary fixes the schedule in advance, the expected number of processes that read $\perp$ is given by a geometric distribution, and is at most $1/\pi_r = n_r^{1/2}$, giving an expected total of at most $2n_r^{1/2}$ processes entering round $r+1$. (The symmetry between the two cases explains the choice of π_r given n_r.) We now compute a high-probability bound for the number of surviving processes.

Let R count the number of processes that read $\perp$. For R to exceed some value m, the first m processes to access the register in round r must choose to read it, which occurs with probability $(1-\pi_r)^m \leq e^{-m\pi_r}$. It follows that $\Pr\left[R \geq (c' \ln n) n_r^{1/2}\right] \leq e^{-c' \ln n} = n^{-c'}$.

Let W be the number of processes that write the register in round $r+1$. Using standard Chernoff bounds (e.g., [12, Theorem 4.4]), we have $\Pr\left[W \geq (c \ln n) n_r^{1/2}\right] \leq 2^{-(c' \ln n) n_r^{1/2}} \leq n^{-c'}$, provided $c' \ln n \geq 6$, which holds easily for sufficiently large n.

Combining these bounds, with probability at least $1 - 2n^{-c}$ it holds that the number of survivors

$$\begin{aligned} W + R &\leq 2(c' \ln n) n_r^{1/2} \\ &\leq 2(c' \ln n) n^{(1/2)\cdot(2/3)^r} \\ &= \frac{2c' \ln n}{n^{(1/6)\cdot(2/3)^r}} \cdot n^{(2/3)^{r+1}} \\ &= \frac{2c' \ln n}{n_r^{1/6}} \cdot n_{r+1} \\ &\leq \frac{2c' \ln n}{\ln^{7/6} n} \cdot n_{r+1} \\ &\leq n_{r+1}, \end{aligned}$$

provided $\ln^{1/6} n \geq 2c'$, which holds for sufficiently large n.

The probability that the induction fails is bounded by $2n^{-c'} = 2n^{-c-1} = \frac{2}{n} n^{-c}$ per round; taking the union bound over $O(\log \log n)$ rounds gives the claimed probability $o(n^{-c})$.

To complete the proof of performance, first observe that the cost of the sifting phase is $O(\log \log n)$, by the above Lemma. The step complexity cost of a call to RatRace [2] with $O(\log^7 n)$[1] other participants is $O(\log \log n)$, with probability at least $1-(\log n)^{-c}$, and $O(\log n)$ otherwise. Choosing $c \geq 2$ gives an expected extra cost of the bad case of $o(1)$. Summing all these costs, we obtain a bound of $O(\log \log n)$ on the expected step complexity of Sift.

Since, with high probability, at most $\log^7 n$ processes participate in the RatRace instance, by the properties of RatRace, we obtain that the step complexity of the Sift algorithm is $O(\log n)$, with high probability.
We have therefore obtained the following bounds on the step complexity of the algorithm.

Lemma 3 (Step Complexity). *The* Sift *algorithm in Figure 1 runs in expected* $O(\log \log n)$ *steps, and in* $O(\log n)$ *steps, with high probability.*

We can extend this argument to obtain an $O(n)$ bound on the total number of steps that processes may take during a run of the algorithm.

Corollary 1 (Total Complexity). *The* Sift *algorithm has total step complexity* $O(n)$*, with high probability.*

Proof. The proof of Lemma 2 states that, with high probability, at most $n^{(2/3)^r}$ processes continue after sifting round r, for any $1 \leq r \leq \kappa \ln \ln n$. Let $\beta = \lfloor \kappa \ln \ln n \rfloor$.

It then follows that the total number of shared-memory operations performed by processes in the sifting phase is at most $\sum_{i=0}^{\beta} n^{(2/3)^r} + \sum_{i=\beta+1}^{\lceil (5/2) \ln \ln n \rceil} n^{(2/3)^\beta} \leq n + \sum_{i=1}^{(5/2) \ln \ln n} n^{2/3} \leq 2n$, with high probability. Since the total number of operations that $\ln^7 n$ participants may perform in RatRace is $O(\text{polylog}\, n)$, with high probability, the claim follows.

We notice that the processes that return loser without writing to any registers during an execution do not incur any read-after-write (RAW) cost in the execution. The above argument provides upper bounds for the number of such processes.

Corollary 2 (RAW Cost). *The* Sift *algorithm incurs* $O(\sqrt{n})$ *expected total RAW cost. With high probability, the algorithm incurs* $O(n^{2/3} \log n)$ *total RAW cost.*

Finally, we notice that the algorithm is correct even if the adversary is strong (the sifting phase may not eliminate any processes). Its expected step complexity in this case is the same as that of RatRace, i.e. $O(\log n)$.

Corollary 3 (Strong Adversary). *Against a strong adversary, the* Sift *algorithm is correct, and has expected step complexity* $O(\log n)$.

4.4 Multi-use Test-and-Set

We now present a transformation from single-use to multi-use test-and-set implementations. This scheme simplifies the technique presented in [1] for single-writer registers, in a setting where multi-writer registers are available. See Figure 2 for the pseudocode.

[1] Reference [2] presents an upper bound of $O(\log^2 k)$ on the step complexity of RatRace with k participants, with high probability. A straightforward refinement of the analysis improves this bound to $O(\log k)$, with high probability.

```
Shared:
T, a vector of linearizable single-use test-and-set objects
Index, an atomic register, initially 0
Local:
crtWinner, a local register, initially false
procedure Test-and-Set() /* at process $p_i$ */
    $v \leftarrow Index$.read()
    $res \leftarrow T[v]$.test-and-set()
    if $res \leftarrow$ winner then
        $crtWinner_i \leftarrow$ true
    return $res$
procedure ReSet()
    if $crtWinner_i =$ true then
        $Index$.write($Index$.read() + 1)
        $crtWinner_i \leftarrow$ false
```

Fig. 2. The multi-use test-and-set algorithm

Description. Processes share a list T of linearizable single-use test-and-set objects, and an atomic register $Index$. Each process maintains a local flag $crtWinner$ indicating whether it is the winner of of the current test-and-set instance. For the test-and-set operation, each process reads in variable v the $Index$ register and calls the single-use test-and-set instance $T[v]$. The process sets the variable $crtWinner$ if it is the current winner of $T[v]$, and returns the value received from $T[v]$. For the reset operation, the current winner increments the value of $Index$ and resets its $crtWinner$ local variable. (Recall that, by the specification of test-and-set [1], only the winner of an instance may reset it.)

The proof of correctness for the transformation is immediate. The complexity bounds are the same as those of the single-use implementation.

Improvements. The above scheme has the disadvantage that it may allocate an infinite number of single-use test-and-set instances. This can be avoided by allowing the current winner to de-allocate the current instance in the reset procedure, and adding version numbers to shared variables, so that processes executing a de-allocated instance automatically return loser.

5 Lower Bound

We first notice that any test-and-set algorithm against an oblivious adversary has executions with step complexity $\Omega(\log n)$, with probability at least $1/n^c$, for a small constant c. In essence, this implies that any bound $O(C)$ on the step complexity of test-and-set that holds with high probability has to have $C \in \Omega(\log n)$. This bound is a corollary of a result by Attiya and Censor-Hillel [4] on the complexity of randomized consensus against a weak adversary, and is matched by our algorithm.

Lemma 4. *For any read-write test-and-set algorithm A that ensures agreement, there exists a weak adversary such that the probability that A does not terminate after* $\log n$ *steps is at least $1/n^\gamma$, for a small constant γ.*

Proof. Consider a test-and-set algorithm A. Then, in every execution in which only two processes participate, the algorithm can be used to solve consensus. We now employ Theorem 5.3 of [4], which states that, for any consensus algorithm for n processes and f failures, there is a weak adversary and an initial configuration, such that the probability that A does not terminate after $k(n - f)$ steps is at least $(1 - c^k \epsilon_k)/c^k$, where c is a constant, and ϵ_k is a bound on the probability for disagreement.

We fix parameters $k = \log n$, $\epsilon_k = 0$ (since the test-and-set algorithm guarantees agreement), $n = 2$, and $f = 1$. The claim follows, with the parameter $\gamma = \log c$.

Strong Adversary RAW Bound. We now apply the lower bound of Attiya *et al.* [5] on the necessity of expensive read-after-write (RAW) patterns in deterministic object implementations, to the case of randomized read-write algorithms against a strong adversary. We state our lower bound in terms of total RAW cost for a test-and-set object; since test-and-set can be trivially implemented using stronger objects such as consensus, adaptive strong renaming, fetch-and-increment, initialized queues and stacks, this bound applies to randomized implementations of these objects as well.

Lemma 5 (Strong Adversary RAW Bound). *Any read-write test-and-set algorithm that ensures safety and terminates with probability α against a* strong adaptive *adversary has worst-case expected total RAW complexity $\Omega(\alpha n)$.*

Proof (Sketch). Let A be a read-write test-and-set algorithm. We describe an adversarial strategy $S(A)$ that forces each process to perform a read-after-write (RAW) operation in all terminating executions. The adversary schedules process p_1 until it has its first write operation enabled. (Each process has to eventually write in a solo execution, since otherwise we can construct an execution with two winners.) Similarly, it proceeds to schedule each process $p_2, \ldots, p_n$ until each has its first write operation enabled (note that no process has read any register that another process wrote at this point). Let $R_1, R_2, \ldots, R_n$ be the registers that $p_1, p_2, \ldots, p_n$ write to, respectively (these registers are not necessarily distinct).

The adversary then schedules process p_n to write to register R_n. It then schedules process p_n until p_n reads from a register that it did not last write to. This has to occur, since otherwise there exists an execution $\mathcal{E}'$ in which process p_n takes the same steps, and a process $q \neq p_n$ is scheduled solo until completion, right before p_n writes to R_n. Process q has to decide winner in this parallel execution. On the other hand, process p_n cannot distinguish execution $\mathcal{E}'$ from a solo execution, therefore decides winner in $\mathcal{E}'$, violating the *unique winner* property. Since the algorithm guarantees safety in all executions, process p_n has to read from a register it did not last write to, and therefore incurs a RAW in execution $\mathcal{E}$.

Similarly, after process p_n performs its RAW, the adversary schedules process p_{n-1} to perform its write operation. By a similar argument to the one above, if the adversary schedules p_{n-1} until completion, p_{n-1} has to read from a register that it did not write to, and incur a RAW. We proceed identically for processes $p_{n-2}, \ldots, p_1$ to obtain that each process incurred a RAW in execution $\mathcal{E}$ dictated by the strong adversarial scheduler, which concludes this sketch of proof.

Discussion. This linear bound applies to randomized algorithms against a strong adversary. Since, by Corollary 2, the Sift test-and-set algorithm has expected RAW cost $O(\sqrt{n})$ against a weak adversary, these two results suggest that randomization can help reduce some of the inherent RAW cost of implementing shared objects, if the scheduler is assumed to be oblivious.

6 Summary and Future Work

In this paper, we present a linearizable implementation of randomized test-and-set, with $O(\log \log n)$ individual step complexity and $O(n)$ total step complexity, against a weak oblivious adversary. Our algorithm shows an exponential improvement over previous solutions, that considered a strong adaptive adversary. Also, it has the interesting property that its performance degrades gracefully if the adversary is adaptive, as the algorithm is still correct in this case, and has step complexity is $O(\log n)$.

Lower bounds represent one immediate direction of future work. In particular, it is not clear whether better algorithms may exist, either for the oblivious adversary, or for the adaptive one. Also, another direction would be to see whether our sifting technique may be applied in the context of other distributed problems, such as mutual exclusion or cooperative collect.

Acknowledgements. The authors would like to thank Keren Censor-Hillel for useful discussions, and the anonymous reviewers for their feedback.

References

1. 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)
2. Alistarh, D., Attiya, H., Gilbert, S., Giurgiu, A., Guerraoui, R.: Fast randomized test-and-set and renaming. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 94–108. Springer, Heidelberg (2010)
3. Attiya, H., Censor, K.: Tight bounds for asynchronous randomized consensus. J. ACM 55(5), 1–26 (2008)
4. Attiya, H., Censor-Hillel, K.: Lower bounds for randomized consensus under a weak adversary. SIAM J. Comput. 39(8), 3885–3904 (2010)
5. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M.M., Vechev, M.T.: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In: POPL, pp. 487–498 (2011)
6. Aumann, Y.: Efficient asynchronous consensus with the weak adversary scheduler. In: PODC 1997: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 209–218. ACM, New York (1997)
7. Golab, W.M., Hadzilacos, V., Hendler, D., Woelfel, P.: Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In: PODC, pp. 3–12 (2007)

8. Golab, W.M., Hendler, D., Woelfel, P.: An o(1) rmrs leader election algorithm. SIAM J. Comput. 39(7), 2726–2760 (2010)
9. Golab, W.M., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: STOC, pp. 373–382 (2011)
10. Herlihy, M.: Randomized wait-free concurrent objects (extended abstract). In: Proceedings of the Tenth Annual ACM symposium on Principles of Distributed Computing, PODC 1991, pp. 11–21. ACM, New York (1991)
11. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
12. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
13. Peterson, G.L., Fischer, M.J.: Economical solutions for the critical section problem in a distributed system (extended abstract). In: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, STOC 1977, pp. 91–97. ACM, New York (1977)
14. Plotkin, S.: Chapter 4: Sticky bits and universality of consensus. Ph.D. Thesis. MIT (1998)
15. Tromp, J., Vitányi, P.: Randomized two-process wait-free test-and-set. Distrib. Comput. 15(3), 127–135 (2002)

Tight Space Bounds for ℓ-Exclusion

Gadi Taubenfeld

The Interdisciplinary Center, P.O. Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il

Abstract. The simplest deadlock-free algorithm for mutual exclusion requires only one single-writer non-atomic bit per process [4,6,13]. This algorithm is known to be space optimal [5,6]. For over 20 years now it has remained an intriguing open problem whether a similar type of algorithm, which uses only one single-writer bit per process, exists also for ℓ-exclusion for some $\ell \geq 2$.

We resolve this longstanding open problem. For any ℓ and n, we provide a tight space bound on the number of single-writer bits required to solve ℓ-exclusion for n processes. It follows from our results that it is not possible to solve ℓ-exclusion with one single-writer bit per process, for any $\ell \geq 2$.

In an attempt to understand the inherent difference between the space complexity of mutual exclusion and that of ℓ-exclusion for $\ell \geq 2$, we define a weaker version of ℓ-exclusion in which the liveness property is relaxed, and show that, similarly to mutual exclusion, this weaker version can be solve using one single-writer non-atomic bit per process.

Keywords: Mutual Exclusion, ℓ-exclusion, space complexity, tight bounds.

1 Introduction

1.1 Motivation

The *ℓ-exclusion* problem, which is a natural generalization of the mutual exclusion problem, is to design an algorithm which guarantees that up to ℓ processes and no more may simultaneously access identical copies of the same non-sharable resource when there are several competing processes. A solution is required to withstand the slow-down or even the crash (fail by stopping) of up to $\ell - 1$ of the processes. A process that fails by crashing simply stops executing more steps of its program, and hence, there is no way to distinguish a crashed process from a correct process that is running very slowly. For $\ell = 1$, the 1-exclusion problem is the familiar mutual exclusion problem.

A good example, which demonstrates why a solution for mutual exclusion does not also solves ℓ-exclusion (for $\ell \geq 2$), is that of a bank where people are waiting for a teller. Here the processes are the people, the resources are the tellers, and the parameter ℓ is to the number of tellers. We notice that the usual bank solution, where people line up in a single queue, and the person at the head of the queue goes to any free teller, does *not* solve the ℓ-exclusion problem. If $\ell \geq 2$ tellers are free, a proper solution should enable the first ℓ people in line to move simultaneously to a teller. However, the bank solution, requires them to move past the head of the queue one at a time. Moreover, if the person at the front of the line "fails", then the people behind this person wait forever. Thus, a better solution is required which will not let a single failure to tie up all the resources.

The simplest deadlock-free algorithm for mutual exclusion, called the One-bit algorithm, requires only one single-writer non-atomic shared bit per process [4,6,13]. The One-bit algorithm is known to be space optimal [5,6]. For over 20 years now it has remained an intriguing open problem whether a similar type of algorithm, which uses only one single-writer bit per process, exists for ℓ-exclusion for some $\ell \geq 2$. In [16], Peterson refers to the One-bit algorithm, and writes: "Unfortunately, there seems to be no obvious generalization of their algorithm to ℓ-exclusion in general". He further points out that it is an interesting open question whether this can be done even for $n = 3$ and $\ell = 2$, where n is the number of processes. This problem is one of the oldest longstanding open problem in concurrent computing.

In this paper we resolve this longstanding open problem. For any ℓ and n, we provide a tight space bound on the number of single-writer bits required to solved the ℓ-exclusion problem for n processes. It follows from our results that only in the case where $\ell = 1$, it is possible to solve the problem with one single-writer bit per process.

1.2 The ℓ-Exclusion Problem

To illustrate the ℓ-exclusion problem, consider the case of buying a ticket for a bus ride. Here a resource is a seat on the bus, and the parameter ℓ is the number of available seats. In the ℓ-exclusion problem, a passenger needs only to make sure that there is some free seat on the bus, but not to reserve a particular seat. A stronger version, called ℓ-assignment (or slotted ℓ-exclusion), would require also to reserve a particular seat.

More formally, it is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section* and *exit*. The ℓ-exclusion problem is to write the code for the *entry code* and the *exit code* in such a way that the following basic requirements are satisfied.

ℓ-Exclusion: *No more than ℓ processes are at their critical sections at the same time.*

ℓ-Deadlock-Freedom: *If strictly fewer than ℓ processes fail and a non-faulty process is trying to enter its critical section, then some non-faulty process eventually enters its critical section.*

We notice that the above standard definition of the ℓ-deadlock-freedom requirement is (slightly) stronger than only requiring that "if fewer than ℓ processes are in their critical sections, then it is possible for another process to enter its critical section, even though no process leaves its critical section in the meantime".

The ℓ-deadlock-freedom requirement may still allow "starvation" of individual processes. It is possible to consider stronger progress requirements which do not allow starvation. In the sequel, by an ℓ-exclusion algorithm we mean an algorithm that satisfies both ℓ-exclusion and ℓ-deadlock-freedom. We also make the standard requirement that the exit code is required to be *wait-free*: once a non-faulty process starts executing its exit code, it always finishes it regardless of the activity of the other processes.

In an attempt to pinpoint the reason for the inherent difference between the space complexity of mutual exclusion and that of ℓ-exclusion for $\ell \geq 2$, we will also consider a weaker version of the ℓ-exclusion problem in which the liveness property is relaxed. Let n be the number of processes,

Weak ℓ-Deadlock-Freedom: *If strictly fewer than ℓ processes fail, at least one non-faulty process is trying to enter its critical section, and at least $n - \ell$ processes are in their remainders, then some non-faulty process eventually enters its critical section, provided that no process leaves its remainder in the meantime.*

The weak ℓ-deadlock-freedom property guarantees that as long as no more than ℓ processes try to enter their critical sections, all non-faulty processes should succeed regardless of how many processes crash. By a *weak* ℓ-exclusion algorithm we mean an algorithm that satisfies (1) ℓ-exclusion, (2) 1-deadlock-freedom, and (3) weak ℓ-deadlock-freedom. For $\ell = 1$, a weak 1-exclusion algorithm is a mutual exclusion algorithm.

1.3 Results

Our model of computation consists of an asynchronous collection of n processes that communicate only by reading and writing *single-writer* registers. A single-writer register can be written by one predefined process and can be read by all the processes. A register can be atomic or non-atomic. With an atomic register, it is assumed that operations on the register occur in some definite order. That is, reading or writing an atomic register is an indivisible action. When reading or writing a non-atomic register, a process may be reading a register while another is writing into it, and in that event, the value returned to the reader is arbitrary [12]. Our results are:

A Space Lower Bound. For any $\ell \geq 2$ and $n > \ell$, any ℓ-exclusion algorithm for n processes must use at least $2n - 2$ bits: at least two bits per process for $n - 2$ of the processes and at least one bit per process for the remaining two processes. (Here a bit can be atomic or non-atomic.)

A Matching Space Upper Bound. For $\ell \geq 2$ and $n > \ell$, there is an ℓ-exclusion algorithm for n processes that uses $2n - 2$ non-atomic bits: two bits per process for $n - 2$ of the processes and one bit per process for the remaining two processes.

An Optimal Weak ℓ-Exclusion Algorithm. For $\ell \geq 2$ and $n > \ell$, there is a weak ℓ-exclusion algorithm for n processes that uses one non-atomic bit per process.

1.4 Related Work

To place our results in perspective, we give a brief history of the ℓ-exclusion problem. The mutual exclusion problem was first stated and solved for n processes by Dijkstra in [7]. Numerous solutions for the problem have been proposed since it was first introduced in 1965. In [5,6], Burns and Lynch have shown that any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers, even when *multi-writer* registers are allowed. This important lower bound can be easily generalized to show that also any ℓ-exclusion algorithm for n processes must use at least n

shared registers for any $\ell \geq 1$. The One-bit mutual exclusion algorithm, which uses n non-atomic bits and hence provides a tight space upper bound, was developed independently by Burns [4] (also appeared in [6]), and by Lamport [13].

The ℓ-exclusion problem, which generalizes the mutual exclusion problem, was first defined and solved in [9,10]. For a model which supports read-modify-write registers, a tight space bound of $\Theta(n^2)$ shared states is proved in [9,10], for the ℓ-exclusion problem *for fixed* ℓ assuming the strong FIFO-enabling liveness property (and strong robustness). There is a large gap between the constants in the upper and lower bounds. Both depend on ℓ, but the constant in the upper bound is exponential in ℓ, while the constant in the lower bound is linear in ℓ. Without the strong liveness property and when not requiring that the exit code be wait-free, $O(n)$ states suffice for mutual exclusion using read-modify-write registers [15]. Several algorithms for ℓ-exclusion, which are based on strong primitives such as fetch-and-increment and compare-and-swap, are considered in [3]. The "bank example" in Section 1.1 is from [10].

In [16], Peterson has proposed several ℓ-exclusion algorithms for solving the problem using atomic read/write registers satisfying various progress properties ranging from ℓ-deadlock-freedom to FIFO-enabling. The simplest algorithm in [16] requires a single 3-valued single-writer atomic register per process. A FIFO-enabling ℓ-exclusion algorithm using atomic registers is presented also in [1], which makes use of concurrent time-stamps for solving the problem with bounded size memory. Long-lived and adaptive algorithms for collecting information using atomic registers are presented in [2]. The authors employ these algorithms to transform the ℓ-exclusion algorithm of [1], into its corresponding adaptive long-lived version. An ℓ-exclusion algorithm using $O(n^2)$ non-atomic single-writer bits which does not permits individual starvation of processes, is presented in [8]. Many known mutual exclusion, ℓ-exclusion and ℓ-assignment algorithms are discussed in details in [17].

2 A Space Lower Bound

In this section we assume a model where the only shared objects are atomic registers. It is obvious that the space lower bound applies also for non-atomic registers. In the following, by a *register* (*bit*), we mean an atomic single-writer register (bit).

Theorem 1. *For any $\ell \geq 2$ and $n > \ell$, any ℓ-exclusion algorithm for n processes must use at least $2n - 2$ bits: at least two bits per process for $n - 2$ of the processes and at least one bit per process for the remaining two processes.*

In the next section we will provide a matching upper bound. Interestingly, Theorem 1 follows from the following special case when $\ell = 2$ and $n = 3$.

Theorem 2. *Any 2-exclusion algorithm for 3 processes must use at least 4 bits: at least two bits for one of the processes and at least one bit for each one of the remaining two processes.*

Proof of Theorem 1: We observe that any ℓ-exclusion algorithm, say A, where $\ell \geq 2$ for processes $p_1, ..., p_n$ (where $n > \ell$), can be transformed into a 2-exclusion algorithm, say A', for processes p_1, p_2, p_3, where the space for each one of the three processes in

A' is the same as the space for the corresponding process in A. Let the $\ell - 2$ processes $p_4, ..., p_{\ell+1}$ execute A until they all crash in their critical sections, and let st be the global state immediately after these $\ell - 2$ processes crash. A' is now constructed from A by replacing each one of the single-writer registers of processes $p_4, ..., p_n$, with a constant its value equals to the value of the corresponding register in state st. The codes of processes p_1, p_2, p_3 in A' are the same as in A, except the fact that in A' whenever a process needs to read the value of a register of one of the processes $p_4, ..., p_n$, it does so by accessing the corresponding constant.

It follows from Theorem 2 and the above transformation that, for *any three* processes from the set of n processes which participate in A, one of three processes must "own" at least two bits and each one of remaining two processes must "own" at least one bit. Thus, the n processes together must use at least $2n - 2$ bits: two bits per process for $n - 2$ of the processes and one bit per process for the remaining two processes. □

For the rest of the section, we focus on proving Theorem 2. The proof is by contradiction. We show that any 2-exclusion algorithm for 3 processes which uses only one single-writer bit per process must have an infinite run in which at least two processes participate infinitely often and where no process ever enters its critical section. This violates the 2-deadlock-freedom requirement.

2.1 The Model

Our model of computation, for proving the lower bound, consists of an asynchronous collection of n processes that communicate via single-writer atomic registers. An *event* corresponds to an atomic step performed by a process. 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 do not return a value. A (global) *state* of an algorithm is completely described by the values of the registers and the values of the location counters of all the processes.

A *run* is a sequence of alternating states and events (also referred to as steps). For the purpose of the lower bound proof, it is more convenient to define a run as a sequence of events omitting all the states except the initial state. Since the states in a run are uniquely determined by the events and the initial state, no information is lost by omitting the states. Each event in a run is associated with a process that is *involved* in the event.

We will use x, y and z to denote runs. The notation $x \leq y$ means that x is a prefix of y, and $x < y$ means that x is a proper prefix of y. When $x \leq y$, we denote by $(y - x)$ the suffix of y obtained by removing x from y. Also, we denote by $x; seq$ the sequence obtained by extending x with the sequence of events seq. We will often use statements like "in run x process p is in its remainder", and implicitly assumed that there is a function which for any finite run and process, lets us know whether a process is in its remainder, entry, critical section, or exit code, at the end of that run. Also, saying that an extension y of x involves only processes from the set P means that all events in $(y - x)$ are only by processes in P. Next we define the *looks like* relation which captures when two runs are indistinguishable to a given process.

Definition 1. *Run x **looks like** run y to process p, if the subsequence of all events by p in x is the same as in y, and the values of all the registers in x are the same as in y.*

The looks like relation is an equivalence relation. The next step by a process always depends on the previous step taken by the process and the current values of the registers. It should be clear that if two runs look alike to a given process then the next step by this process in both runs is the same.

Lemma 1. *Let x be a run which looks like run y to every process in a set P. If z is an extension of x which involves only processes in P then $y; (z - x)$ is a run.*

Proof. By a simple induction on k – the number of events in $(z - x)$. The basis when $k = 0$ holds trivially. We assume that the Lemma holds for $k \geq 0$ and prove for $k + 1$. Assume that the number of events in $(z - x)$ is $k + 1$. For some event e, it is the case that $z = z'; e$. Since the number of events in $(z' - x)$ is k, by the induction hypothesis $y' = y; (z' - x)$ is a run. Let $p \in P$ be the process which involves in e. Then, from the construction, the runs z' and y' look alike to p, which implies that the next step by p in both runs is the same. Thus, since $z = z'; e$ is a run, also $y'; e = y; (z - x)$ is a run. □

To prove Theorem 2, we assume to the contrary that there exists a 2-exclusion algorithm, called $2EX$, for three processes which uses only one single-writer bit per process and show that this assumption leads to a contradiction. All the definitions and lemmas below refer to this arbitrary 2-exclusion algorithm for *three* processes.

2.2 Changing the Value of a Bit

For process p and run x, we use the notation $value(x, p)$ to denote the value in x of the single-writer bit that only p is allowed to write into. I.e., the value of p's single-writer bit at the global state immediately after the last event of x has occurred.

Lemma 2. *Let P be a set of processes where $|P| \leq 2$, let z be a run in which the processes in P are in their critical sections, and let x be the longest prefix of z such that the processes in P are in their remainder regions in x. If $(z - x)$ involves only steps by processes in P, then $value(x, p) \neq value(z, p)$ for every $p \in P$.*

Proof. Assume to the contrary that $value(x, p) = value(z, p)$ for some $p \in P$.

Case 1: $|P| = 1 = \{p\}$. Since non of the events in $(z - x)$ involves the other two processes, x looks like z to all processes other than p. By the 2-deadlock-freedom property, there is an extension of x which does not involve p in which the other two processes enter their critical sections. Since x looks like z to all processes other than p, by Lemma 1, a similar extension exists starting from z. That is, the other two processes can enter their critical sections in an extension of z, while p is still in its critical section. This violates the 2-exclusion property.

Case 2: $|P| = 2 = \{p, q\}$, $value(x, p) = value(z, p)$ and $value(x, q) = value(z, q)$. Since non of the events in $(z - x)$ involves the third process, call it process r, x looks like z to r. By the 2-deadlock-freedom property, there is an extension of x which involve only r in which r enters its critical section. Since x looks like z to r, by Lemma 1, a similar extension exists starting from z. That is, r can enter its critical section in an extension of z, while p and q are still in their critical sections. This violates the 2-exclusion property.

Case 3: $|P| = 2 = \{p, q\}$, $value(x, p) = value(z, p)$ and $value(x, q) \neq value(z, q)$. By the 2-deadlock-free property, there is an extension y of x in which q is in its critical section and $(y - x)$ involves only q. By case 1 above, $value(x, q) \neq value(y, q)$. Since non of the events in $(y - x)$ and in $(z - x)$ involves r, y looks like z to r. By the 2-deadlock-freedom property, there is an extension of y which involve only r in which r enters its critical section. Since y looks like z to r, by Lemma 1, a similar extension exists starting from z. That is, r can enter its critical section in an extension of z, while p and q are still in their critical sections. This violates the 2-exclusion property. □

Lemma 3. *Let x be a run in which p is in its remainder and let z be the longest prefix of x such that p is in its critical section in z. If $(x - z)$ involves only steps by p, then $value(x, p) \neq value(z, p)$.*

Proof. Assume to the contrary that $value(x, p) = value(z, p)$. Since non of the events in $(x - z)$ involves the other processes, x looks like z to all processes other than p. By the 2-deadlock-freedom property, there is an extension of x which does not involve p in which the other two processes enter their critical sections. Since x looks like z to all the processes other than p, by Lemma 1, a similar extension exists starting from z. That is, the other two processes can enter their critical sections in an extension of z, while p is still in its critical section. This violates the 2-exclusion property. □

2.3 Locking

Next we introduce the key concept of a *locked* process. Intuitively, process p is *locked* in a given run, if p *must* wait for at least one other process to take a step before it may enter its critical section.

Definition 2. *For process p and run x, p is **locked** in x, if (1) p is in its entry code in x, and (2) for every extension y of x such that $(y - x)$ involves only steps by p, p is in its entry code in y.*

Lemma 4. *There exists a run x_0, such that all three processes are locked in x_0, and no process is in its critical section in any prefix of x_0.*

Proof. The run x_0 is constructed as follows. We start from (any) one of the possible initial states, and denote the initial values of the three bits of the processes p_1, p_2, p_3 in the initial state by $init(p_1)$, $init(p_2)$ and $init(p_3)$, respectively. We first let p_1 run alone until it is about to write and change the value of its single-writer bit from $init(p_1)$ to $1 - init(p_1)$ for the *last* time before it may enter its critical section if it continues to run alone (by Lemma 2 such a write must eventually happen). We suspend p_1 just before it writes and repeat this procedure with p_2 and then with p_3. Then we let the processes p_1, p_2, p_3 to write the values $1 - init(p_1), 1 - init(p_2), 1 - init(p_3)$, respectively, into their bits. The resulting run, where all three bits are set to values which are different from their initial values, is x_0. Each process $p \in \{p_1, p_2, p_3\}$, can not distinguish x_0 from a run in which before p has executed this last write, the other two processes run alone, flipped the values of their bits (by Lemma 2), and have entered their critical sections. Thus, it follows from the 2-exclusion property and Lemma 1, that there can not be an extension of x_0 by steps of p only in which p is in its critical section, which implies that each one of the three processes is locked in x_0. □

Lemma 5. *Assume that x looks like z to p. Process p is locked in x if and only if p is locked in z.*

Proof. Assume to the contrary that p is locked in x and p is *not* locked in z. By definition, there is an extension $\hat{z}$ of z such that $(\hat{z} - z)$ involves only p and p is in its critical section in $\hat{z}$. Since x looks like z to p, by Lemma 1, a similar extension exists starting from x. That is, p is in its critical section in $x; (\hat{z} - z)$. This contradicts the assumption the p is locked in x. Since the *looks like* relation is symmetric, the result follows. □

Lemma 6. *Assume that p is in its remainder in x. For every $q \neq p$, q is not locked in x.*

Proof. Immediately from the 2-deadlock-freedom requirement. □

2.4 Locking and Values

The following lemmas relate between locked processes and the values of their bits.

Lemma 7. *Assume that the three processes are locked in x, and let z be an extension of x such that p is not involved in $(z-x)$ and q is in its remainder in z. Then, $value(x, q) \neq value(z, q)$.*

Proof. Assume to the contrary that $value(x, q) = value(z, q)$. Let r be the third process. By the 2-deadlock-freedom requirement, there is an extension $\hat{z}$ of z such that (1) $(\hat{z} - z)$ involves only r, and (2) r is in its critical section in $\hat{z}$. There are two cases.

Case 1: $value(x, r) = value(\hat{z}, r)$. In this case, x looks like $\hat{z}$ to p. By Lemma 5, p is locked in $\hat{z}$. This violates lemma 6 (i.e., this violates the 2-exclusion property).

Case 2: $value(x, r) \neq value(\hat{z}, r)$. By lemma 3 and the assumption that the exit code is wait-free, there is an extension z' of $\hat{z}$, such that (1) $(z' - \hat{z})$ involves only r, r is in its remainder in z' and $value(\hat{z}, r) \neq value(z', r)$. Thus, $value(x, r) = value(z', r)$. This implies that x looks like z' to p. By Lemma 5, p is locked in z'. This violates Lemma 6. □

Lemma 8. *Assume that the three processes are locked in x, and let z be an extension of x such that p is not involved in $(z - x)$ and q is in its critical section in z. Then, $value(x, q) = value(z, q)$.*

Proof. By lemma 3 and the assumption that the exit code is wait-free, there is an extension z' of z, such that (1) $(z' - z)$ involves only q, q is in its remainder in z' and $value(z, q) \neq value(z', q)$. By Lemma 7, $value(x, q) \neq value(z', q)$ Thus, $value(x, q) = value(z, q)$. □

Lemma 9. *Assume that the three processes are locked in x. For every two processes p and q there is an extension z of x such that:* (1) $(z - x)$ *involves only p and q,* (2) *p is in its critical section in z,* (3) *q is in its remainder is z,* (4) $value(x, p) = value(z, p)$, *and* (5) $value(x, q) \neq value(z, q)$.

Proof. We construct z as follows. Starting from x we first let p and q run until one of them enters its critical section. This is possible by the 2-deadlock-freedom requirement. Then, we let the process that has entered its critical section, continue alone until it reaches its remainder regions (recall that the exit code is assumed to be wait-free). At that point we let the other process run alone until it enters its critical section (again, this is possible by the 2-deadlock-freedom requirement), and we let this process continues to run until it also reaches its remainder region. At this point both p and q are in their remainders. Now we let, p run until it enters its critical section (again, this is possible by the 2-deadlock-freedom requirement). The resulting run, where p is in its critical section, and process q is in its remainder is z. By Lemma 8, $value(x, p) = value(z, p)$, and by Lemma 7, $value(x, q) \neq value(z, q)$. □

2.5 Flexibility

Intuitively, process p is *flexible* in a given run, if p *can* change the value of its bit without a need to wait for some other process to take a step.

Definition 3. *For process p and run x, p is* ***flexible*** *in x, if there exists an extension z of x such that: (1) $(z - x)$ involves only steps by p, (2) for every $x \leq y \leq z$, p is in its entry code in y, and (3) $value(x, p) \neq value(z, p)$.*

Lemma 10. *Let x be a run such that the three processes are locked in x. Then, at least two processes are flexible in x.*

Proof. We assume that p_1, p_2 and p_3 are locked in x. By the 2-deadlock-freedom property, if we extend x by letting processes p_1 and p_2 taking steps alternately, eventually one of the two must enter its critical section. Let z be the shortest extension of x such that in z both processes are still in their entry code, but the next step of one of them is already in a critical section.

Case 1: for some y where $x \leq y \leq z$, either $value(x, p_1) \neq value(y, p_1)$ or $value(x, p_2) \neq value(y, p_2)$. Let y be the *shortest* run where $x < y \leq z$, and $value(x, p_1) \neq value(z, p_1)$ or $value(x, p_2) \neq value(z, p_2)$. This means that only one process, say p_1, changed its bit in y, and all the steps of p_2 in $(y - x)$ are invisible to (i.e., do not involve) p_1 and p_3. Thus, it is possible to remove from y all the events in which p_2 is involved in $(y - x)$ and get a new run y_1. Clearly y_1 looks like y to p_1 and hence $value(y, p_1) = value(y_1, p_1)$, which implies that $value(x, p_1) \neq value(y_1, p_1)$. Thus, y_1 is a *witness* for the fact that p_1 is flexible in x.

Case 2: for all y where $x \leq y \leq z$, $value(x, p_1) = value(y, p_1)$ and $value(x, p_2) = value(y, p_2)$. Thus, all the steps of the p_1 in $(z - x)$ are invisible to p_2 and p_3, and all the steps of the p_2 in $(z - x)$ are invisible to p_1 and p_3. Thus, it is possible to remove from z all the events in which p_2 is involved in $(z - x)$ and get a new run z_1, and similarly, it is possible to remove from z all the events in which p_1 is involved in $(z - x)$ and get a new run z_2. Clearly, z_1 looks like z to p_1 and z_2 looks like z to p_2. By definition, either the next step of p_1 from z_1 is already a step in its critical section, or the next step of p_2 from z_2 is already a step in its critical section. This contradicts the assumption that all the processes are locked in x.

We conclude that either p_1 or p_2 is flexible in x. Assume w.l.o.g. that p_1 is flexible in x. We repeat the above argument to show that either p_2 or p_3 is flexible in x. □

2.6 Main Lemma and Proof of Theorem 2

We are now ready to prove the main lemma and complete the proof of Theorem 2.

Lemma 11 (main lemma). *Let x be a run such that all three processes are locked in x. Then, there exists an extension z of x such that (1) all the processes are locked in z, (2) two of the processes are involved in $(z - x)$, and (3) for every $x \leq y \leq z$, all the processes are in their entry codes in y.*

Proof. We will construct the run z and show that z satisfies the required properties. We start from the run x in which the three processes, p_1, p_2, p_3 are locked. By Lemma 10 at least two processes are flexible in x. W.l.o.g. we assume that p_2 and p_3 are flexible in x. We construct and extension of x, called x_2, which involves only process p_2 as follows. Starting from x, we let p_2 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_2)$ to $1 - value(x, p_2)$. We suspend p_2 just *before* it writes, and call this run x_2. For later reference, we denote by e_{p_2} the write event that p_2 is about to take once activated again, and denote by x'_2 the run $x_2; e_{p_2}$.

Similarly, we construct and extension of x, called x_3, which involves only process p_3 as follows. Starting from x, we let p_3 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_3)$ to $1 - value(x, p_3)$. We suspend p_3 just *before* it writes, and call this run x_3. For later reference, we denote by e_{p_3} the write event that p_3 is about to take once activated again, and denote by x'_3 the run $x_3; e_{p_3}$.

Next we construct an extension of x, called x_{23}, which involves only p_2 and p_3. We let p_2 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_2)$ to $1 - value(x, p_2)$. We suspend p_2 just before it writes and then let p_3 run alone until it is about to write and change the value of its single-writer bit from $value(x, p_3)$ to $1 - value(x, p_3)$. We suspend p_3 just before it writes. Then we let the processes p_2 and p_3 to write the values $1 - value(x, p_2)$ and $1 - value(x, p_3)$ respectively, into their bits in an arbitrary order. The resulting run, where the bits of p_2 and p_3 are set to values which are different from their values in x, is run x_{23}. Thus, the run x_{23} is the run $x_2; (x_3 - x); e_{p_2}; e_{p_3}$ (the order in which e_{p_2} and e_{p_3} are executed is immaterial). We observe that $x_2; (x_3 - x)$ looks like $x_3; (x_2 - x)$ to all the processes.

We consider also the following two extension of x. By Lemma 9, there exists an extension of x which we will call run x_{12}, such that $(x_{12} - x)$ involves only p_1 and p_2, p_1 is in its critical section in x_{12}, p_2 is in its remainder in x_{12}, $value(x, p_1) = value(x_{12}, p_1)$, and $value(x, p_2) \neq value(x_{12}, p_2)$. By Lemma 9, there exists another extension of x which we will call run x_{13}, such that $(x_{13} - x)$ involves only p_1 and p_3, p_1 is in its critical section in x_{13}, p_3 is in its remainder in x_{13}, $value(x, p_1) = value(x_{13}, p_1)$, and $value(x, p_3) \neq value(x_{13}, p_3)$.

Let the run $\widehat{x_{12}}$ be the run $x; (x_3 - x); (x_{12} - x); e_{p_3}$, and let the run $\widehat{x_{13}}$ be the run $x; (x_2 - x); (x_{13} - x); e_{p_2}$. Clearly, x_{23} looks like $\widehat{x_{12}}$ to p_3, and x_{23} looks like $\widehat{x_{13}}$ to p_3. Informally, this implies that in x_{23}, process p_2 "suspects" that p_1 is in its critical section and p_3 is in its remainder; and process p_3 "suspects" that p_1 is in its critical section and p_2 is in its remainder. Since p_2 is in its remainder in $\widehat{x_{12}}$, by Lemma 6, p_3 is not locked in $\widehat{x_{12}}$, and hence by Lemma 5, p_3 is also not locked in x_{23}. Similarly, since p_3 is in its remainder in $\widehat{x_{13}}$, by Lemma 6, p_2 is not locked in $\widehat{x_{13}}$, and hence by Lemma 5, p_2 is also not locked in x_{23}.

Since both p_2 and p_3 are not locked in x_{23}, there is an extension y_2 of x_{23} by steps of p_2 only in which p_2 is in its critical section, and there is (another) extension y_3 of x_{23} by steps of p_3 only in which p_3 is in its critical section. Since both $(y_2 - x)$ and $(y_3 - x)$ do not involve p_1, by Lemma 8, $value(x, p_2) = value(y_2, p_2)$ and $value(x, p_3) = value(y_3, p_3)$. Thus, $value(x_{23}, p_2) \neq value(y_2, p_2)$ and $value(x_{23}, p_3) \neq value(y_3, p_3)$. This means that in $(y_2 - x_{23})$ there is a write event by p_2, denoted $e^{z_2}_{p_2}$, which changes the value of the bit of p_2 to $value(x, p_2)$, and in $(y_3 - x_{23})$ there is a write event by p_3, denoted $e^{z_3}_{p_3}$, which changes the value of the bit of p_3 to $value(x, p_3)$. Let run z_2 be the shortest extension of x_{23} such that $z_2; e^{z_2}_{p_2}$ is a prefix of y_2, and let run z_3 be the shortest extension of x_{23} such that $z_3; e^{z_3}_{p_3}$ is a prefix of y_2.

Next we construct the extension z of x_{23} which involves only p_2 and p_3. We first let p_2 run alone until it is about to change the value of its single-writer bit to $value(x, p_2)$. We suspend p_2 just before it writes and then let p_3 run alone until it is about to change the value of its single-writer bit to $value(x, p_3)$. We suspend p_3 just before it writes. Then we let the processes p_2 and p_3 to write the values $value(x, p_2)$ and $value(x, p_3)$ respectively, into their bits in an arbitrary order. The resulting run, where the bits of p_2 and p_3 are set to values which are the same as their values in x, is the run z. That is, z is exactly the run $z_2; (z_3 - x_{23}); e^{z_2}_{p_2}; e^{z_3}_{p_3}$ (the order in which the last two write events are executed is immaterial). Below we prove that the three processes are locked in z.

Since $(z - x)$ does not involve p_1, x looks like z to p_1, and thus by Lemma 5, p_1 is locked in z. To prove that the also p_2 and p_3 are locked in x, we will show that each one of them can not distinguish between z and another run in which the other two processes are in their critical sections. We now construct these two runs.

By Lemma 9, there exists an extension of x which we will call run w, such that $(w - x)$ involves only p_2 and p_3, p_3 is in its critical section in w, p_2 is in its remainder in w, $value(x, p_3) = value(w, p_3)$, and $value(x, p_2) \neq value(w, p_2)$. Clearly w looks like x'_2 to p_1. Since p_2 is in its remainder in w, by Lemma 6, p_1 is not locked in w, and hence by Lemma 5, p_1 is also not locked in x'_2. Thus, there is an extension y_1 of x'_2 by steps of p_1 only, in which p_1 is in its critical section. Since $x'_2 - x$ does not involve steps by p_1, $value(x, p_1) = value(x'_2, p_1)$. Since $(y_1 - x)$ does not involve steps by p_3, by Lemma 8, $value(x, p_1) = value(y_1, p_1)$. Thus also $value(x'_2, p_1) = value(y_1, p_1)$.

Recall that $x_{23} = x_2; (x_3 - x); e_{p_2}; e_{p_3}$. We observe that (1) $(x_3 - x)$ involves only p_3 and does not change the value of the bit of p_3, and (2) $(y_1 - x'_2)$ involves only p_1 and does not change the value of the bit of p_1. Thus, by Lemma 1 (applied several times), the sequence of events, $x'_{23} = x_2; (x_3 - x); e_{p_2}; (y_1 - x'_2); e_{p_3}$ is a legal run in which p_1 is in its critical section.

Since x'_{23} looks like x_{23} to p_2 and to p_3, by Lemma 1, $x'_{23}; (y_3 - x_{23})$ is a run in which both p_1 and p_3 are in their critical sections, and $x'_{23}; (z_2 - x_{23})$ is a legal run. Since $(z_2 - x_{23})$ involves only p_2 and the value of the bit of p_2 does not change, by Lemma 1, the sequence $y'_3 = x'_{23}; (z_2 - x_{23}); (y_3 - x_{23}); e^{z_2}_{p_2}$ is a legal run in which both p_1 and p_3 are in their critical sections. Process p_2, can not distinguish z from a run in which before p_2 executed its last write in z, the other two processes have entered their critical sections. That is, z looks like y_3' to p_2. Thus, it follows from the 2-exclusion property and Lemma 1, that there can not be an extension of z by steps of p_2 only in which p_2 is in its critical section. This implies that p_2 is locked in z.

Similarly, since x_{23}' looks like x_{23} to p_2 and to p_3, by Lemma 1, $x_{23}'; (y_2 - x_{23})$ is a run in which both p_1 and p_2 are in their critical sections, and $x_{23}'; (z_3 - x_{23})$ is a legal run. Since $(z_3 - x_{23})$ involves only p_3 and the value of the bit of p_3 does not change, by Lemma 1, the sequence $y_2' = x_{23}'; (z_3 - x_{23}); (y_2 - x_{23}); e_{p_3}^{z_3}$ is a legal run in which both p_1 and p_2 are in their critical sections. Process p_3, can not distinguish z from a run in which before p_3 executed its last write in z, the other two processes have entered their critical sections. That is, z looks like y_2' to p_3. Thus, it follows from the 2-exclusion property and Lemma 1, that there can not be an extension of z by steps of p_3 only in which p_3 is in its critical section. This implies that p_3 is locked in z.

To conclude, we have shown that all three processes are locked in z. Furthermore, it follows from the construction that two of the processes, p_2 and p_3, are involved in $(z - x)$, and that for every $x \leq y \leq z$, all the processes are in their entry codes in y. □

Proof of Theorem 2: We have assumed to the contrary that $2EX$ is a 2-exclusion algorithm for 3 processes which uses one single-writer bit per process. We show that this leads to a contradiction. Starting from x_0 found in Lemma 4, we repeatedly apply the result of Lemma 11, to construct the desired infinite run, in which at least two processes take infinitely many steps, but no process ever enters its critical section. That is, we begin with x_0 and pursue the following *locking-preserving scheduling* discipline:

```
1 $x := x_0$;                                          /* initialization */
2 repeat
3    let $z$ be an extension $x$, its existence is proved in Lemma 11, where all the
     processes are locked in $z$, two processes are involved in $(z - x)$, and
     for every $x \leq y \leq z$, all the processes are in their entry codes in $y$.
4    $x := z$                              /* "locking extension" of $x$ */
5 forever
```

The above scheduling discipline, implies that there is an infinite run of $2EX$ in which at least two processes take infinitely many steps, but no process ever enters its critical section. The existence of such a run implies that $2EX$ does not satisfy 2-deadlock-freedom. A contradiction. □

3 A Space Upper Bound: The Two-bits Algorithm

In this section we provide a tight space upper bound for ℓ-exclusion. To make the upper bound as strong as possible, we will assume that the registers are non-atomic.

Theorem 3. *For $\ell \geq 2$ and $n > \ell$, there is an ℓ-exclusion algorithm for n processes that uses $2n - 2$ non-atomic bits: two bits per process for $n - 2$ of the processes and one bit per process for the remaining two processes.*

We present below a space optimal algorithm which is inspired by Peterson's ℓ-exclusion algorithm [16]. Our algorithm is for n processes each with unique identifier taken from the set $\{1, ..., n\}$. For each process $i \in \{2, ..., n-1\}$,the algorithm requires two single-writer non-atomic bits, called $Flag_1[i]$ and $Flag_2[i]$. For process 1 the algorithm requires one single-writer non-atomic bit, called $Flag_1[1]$, and for process n the algorithm requires one single-writer non-atomic bit, called $Flag_2[n]$. In addition two local variables, called *counter* and j, are used for each process. ℓ is used as a constant.

THE TWO-BITS ℓ-EXCLUSION ALGORITHM: `process` $i \in \{1, ..., n\}$ `program.`

Shared: $Flag_1[1..n-1]$, $Flag_2[2..n]$: arrays of non-atomic bits, initially all entries are 0.
Local: *counter*, j: integer ranges over $\{0, ..., n\}$.
Constant: $Flag_1[n] = 0$, $Flag_2[1] = 0$. /* used for simplifying the presentation */

```
1   if $i \neq n$ then $Flag_1[i] := 1$ fi;                            /* save one bit */
2   repeat
3      repeat
4         $counter := 0$;
5         for $j := 1$ to $n$ do
6            if ($j < i$ and $Flag_1[j] = 1$) or ($Flag_2[j] = 1$)
7            then $counter := counter + 1$ fi od
8      until $counter < \ell$;
9      if $i \neq 1$ then $Flag_2[i] := 1$ fi;                         /* save one bit */
10     $counter := 0$;
11     for $j := 1$ to $n$ do
12        if ($j < i$ and $Flag_1[j] = 1$) or ($j \neq i$ and $Flag_2[j] = 1$)
13        then $counter := counter + 1$ fi od
14     if $counter \geq \ell$ then if $i \neq 1$ then $Flag_2[i] := 0$ fi fi
15  until $counter < \ell$;
16  critical section;
17  if $i \neq 1$ then $Flag_2[i] := 0$ fi; if $i \neq n$ then $Flag_1[i] := 0$ fi;
```

In lines 1, process i (where $i \neq n$) first indicates that it is contending for the critical section by setting $Flag_1[i]$ to 1. Then, in the first repeat loop (lines 3–8) it finds out how many processes have higher priority than itself. A process k has higher priority than process i, if its second flag bit $Flag_2[k]$ is set to 1, or if $k < i$ and $Flag_1[k] = 1$. If less than ℓ processes have higher priority, i exits the repeat loop (line 8). Otherwise, process i waits by spinning in the inner repeat loop (lines 3–87), until less than ℓ processes have higher priority. Once it exits the inner loop it sets its second flag, $Flag_2[i]$, to 1 (for $i \neq 1$). Then, again, it finds out how many processes have higher priority than itself. If less than ℓ processes have higher priority, process i exits the outer repeat loop (line 15) and can safely enter its critical section. Otherwise, the process sets its second flag bit back to 0, and go back to wait in the inner repeat loop (lines 3–8). In the exit code a process simply sets its flag bits to 0. A detailed proof appears in the full version.

The algorithm is also resilient to the *failure by abortion* of any finite number of processes. By an abort-failure of process p, we mean that the program counter of p is set to point to the beginning of its remainder code and that the values of all the single-writer bits of p are set to their initial (default) values. The process may then resume its execution, however, if a process keeps failing infinitely often, then it may prevent other processes from entering their critical sections.

4 Weak ℓ-Exclusion

Recall that a *weak* ℓ-exclusion algorithm is an algorithm that satisfies (1) ℓ-exclusion, (2) 1-deadlock-freedom, and (3) weak ℓ-deadlock-freedom. Next we show that the tight bound for mutual exclusion [5,6], of one bit per process, holds for weak ℓ-exclusion.

Theorem 4. *There is a weak ℓ-exclusion algorithm for n processes which uses one single-writer non-atomic bit per process, for $\ell \geq 1$.*

There may be up to n processes potentially contending to enter their critical sections, each has a unique identifier from the set $\{1, ..., n\}$. The algorithm makes use of a shared array *Flag*, where, for every $1 \leq i \leq n$, all the processes can read the boolean registers $Flag[i]$, but only process i can write $Flag[i]$. ℓ is used as a constant.

ALGORITHM 2. `process` $i \in \{1, ..., n\}$ `program.`

Shared: $Flag[1..n]$: array of non-atomic bits, initially all entries are 0.
Local: $lflag[1..n]$: array of bits; $counter$, j: integer ranges over $\{0, ..., n\}$.

```
1   $counter := 0$;
2   repeat
3       if $counter < \ell$ then $Flag[i] := 1$ fi;
4       $counter := 0$;
5       for $j := 1$ to $i-1$ do $lflag[j] := Flag[j]$; $count := count + lflag[j]$ od;
6       if $counter \geq \ell$ and $Flag[i] = 1$ then $Flag[i] := 0$ fi;
7   until $Flag[i] = 1$;
8   for $j := i+1$ to $n$ do $lflag[j] := Flag[j]$; $counter := counter + lflag[j]$ od;
9   while $counter > \ell$ do
10      for $j := 1$ to $n$ do
11          if $(Flag[j] = 0)$ and $(lflag[j] = 1)$
12          then $lflag[j] := 0$; $counter := counter - 1$ fi
13      od
14  od
15  critical section;
16  $Flag[i] := 0$;
```

In lines 1–7, process i first indicates that it is contending for the critical section by setting its flag bit to 1 (line 3), and then it tries to read the flag bits of all the processes which have identifiers smaller than itself. If less than ℓ of these bits are 0, i exits the repeat loop (line 7). Otherwise, i sets its flag bit to 0, waits until the values of less than ℓ of the flag bits of processes which have identifiers smaller than itself are 0 and starts all over again. In line 8, i reads the flag bits of all the processes which have identifiers greater than itself and remembers their values. Then, in the while loop in lines 9–14, it continuously reads the n flag bits, and it exits the loop only when it finds that at least $n - \ell$ of the flag bits have been 0 at least once since it has set its flag bit to 1. At that point it can safely enter its critical section. A detailed proof appears in the full version.

5 Discussion

For any ℓ and n, we provide a tight space bound on the number of single-writer bits required to solve ℓ-exclusion for n processes. It is easy to modify the two-bits algorithm (from Section 3), so that it uses a single 3-valued single-writer atomic register for $n-2$ of the processes and one bit per process for the remaining two processes. This, together with the result stated in Theorem 1, provides a tight space bound for the size and number of single-writer multi-valued registers required to solve ℓ-exclusion for n processes. We leave open the question of what is the bound for *multi-writer* registers.

References

1. Afek, Y., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: A bounded first-in, first-enabled solution to the ℓ-exclusion problem. ACM Transactions on Programming Languages and Systems 16(3), 939–953 (1994)
2. Afek, Y., Stupp, G., Touitou, D.: Long-lived adaptive collect with applications. In: Proc. 40th IEEE Symp. on Foundations of Computer Science, pp. 262–272 (October 1999)
3. Anderson, J.H., Moir, M.: Using local-spin k-exclusion algorithms to improve wait-free object implementations. Distributed Computing 11 (1997)
4. Burns, J.E.: Mutual exclusion with linear waiting using binary shared variables. SIGACT News 10(2), 42–47 (1978)
5. Burns, J.E., Lynch, A.N.: Mutual exclusion using indivisible reads and writes. In: 18th Annual Allerton Conference on Communication, Control and Computing, pp. 833–842 (1980)
6. Burns, J.N., Lynch, N.A.: Bounds on shared-memory for mutual exclusion. Information and Computation 107(2), 171–184 (1993)
7. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM 8(9), 569 (1965)
8. Dolev, D., Gafni, E., Shavit, N.: Toward a non-atomic era: ℓ-exclusion as a test case. In: Proc. 20th ACM Symp. on Theory of Computing, pp. 78–92 (1988)
9. Fischer, M.J., Lynch, N.A., Burns, J.E., Borodin, A.: Resource allocation with immunity to limited process failure. In: Proc. 20th IEEE Symp. on Foundations of Computer Science, pp. 234–254 (October 1979)
10. Fischer, M.J., Lynch, N.A., Burns, J.E., Borodin, A.: Distributed FIFO allocation of identical resources using small shared space. ACM Trans. on Programming Languages and Systems 11(1), 90–114 (1989)
11. 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)
12. Lamport, L.: On Interprocess Communication, Parts I and II. Distributed Computing 1(2), 77–101 (1986)
13. Lamport, L.: The mutual exclusion problem: Part II – statement and solutions. Journal of the ACM 33, 327–348 (1986)
14. Loui, M.C., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
15. Peterson, G.L.: New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester (February 1980) (Corrected, November 1994)
16. Peterson, G.L.: Observations on ℓ-exclusion. In: 28th Annual Allerton Conference on Communication, Control and Computing, pp. 568–577 (October 1990)
17. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming, 423 pages. Pearson / Prentice-Hall (2006) ISBN 0-131-97259-6

SMV: Selective Multi-Versioning STM

Dmitri Perelman*, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar

Technion, Israel Institute of Technology
{dima39@tx,szaparka@t2,smalish@t2,idish@ee}.technion.ac.il

Abstract. We present *Selective Multi-Versioning (SMV)*, a new STM that reduces the number of aborts, especially those of long read-only transactions. SMV keeps old object versions as long as they might be useful for some transaction to read. It is able to do so while still allowing reading transactions to be invisible by relying on automatic garbage collection to dispose of obsolete versions.

SMV is most suitable for read-dominated workloads, for which it performs better than previous solutions. It has an up to $\times 7$ throughput improvement over a single-version STM and more than a two-fold improvement over an STM keeping a constant number of versions per object. We show that the memory consumption of algorithms keeping a constant number of versions per object might grow exponentially with the number of objects, while SMV operates successfully even in systems with stringent memory constraints.

1 Introduction

Software Transactional Memory (STM) [20,32] is an increasingly popular paradigm for concurrent computing in multi-core architectures. STMs speculatively allow multiple transactions to proceed concurrently, which leads to aborting transactions in some cases. As system load increases, aborts may become frequent, especially in the presence of long-running transactions, and may have a devastating effect on performance [3]. Reducing the number of aborts is thus an important challenge for STMs.

Of particular interest in this context is reducing the abort rate of read-only transactions. Read-only transactions play a significant role in various types of applications, including linearizable data structures with a strong prevalence of read-only operations [21], or client-server applications where an STM infrastructure replaces a traditional DBMS approach (e.g., FenixEDU web application [9]). Particularly long read-only transactions are employed for taking consistent snapshots of dynamically updated systems, which are then used for checkpointing, process replication, monitoring program execution, gathering system statistics, etc.

Unfortunately, long read-only transactions in current leading STMs tend to be repeatedly aborted for arbitrarily long periods of time. As we show below, the time for completing such a transaction varies significantly under contention, to the point that some read-only transactions simply cannot be executed without "stopping the world". As mentioned by Cliff Click [1], this kind of instability is one of the primary practical disadvantages of STM; Click mentions *multi-versioning* [5] (i.e., keeping multiple versions per object), as a promising way to make program performance more predictable.

* This work was partially supported by Hasso Plattner Institute.

Indeed, by keeping multiple versions it is possible to assure that each read-only transaction successfully commits by reading a *consistent snapshot* [4] of the objects it accesses, e.g., values that reflect updates by transactions that committed before it began and no partial updates of concurrent transactions. This way, multiple versions have the potential to improve the performance of single-versioned STMs [19,16,13,11], which, as we show below, might waste as much as 80% of their time because of aborts in benchmarks with many read-only transactions.

However, using multiple versions introduces the challenge of their efficient garbage collection. As we show below, a simple approach of keeping a constant number of versions for each object does not provide enough of a performance benefit, and, even worse, can cause severe memory problems in long executions. We further demonstrate in Section 3 that the memory consumption of algorithms keeping k versions per object might grow exponentially with the number of objects. The challenge is, therefore, to devise an approach for efficient management of old object versions.

In Section 4, we present *Selective Multi-Versioning (SMV)*, a novel STM algorithm that addresses this challenge. SMV keeps old object versions that are still useful to potential readers, and removes ones that are obsolete. This way, read-only transactions can always complete – they neither block nor abort – while for infrequently-updated objects only one version is kept most of the time.

SMV achieves this while allowing read-only transactions to remain *invisible* [13], i.e., having no effect on shared memory. At first glance, combining invisible reads with effective garbage collection may seem impossible — if read-only transactions are invisible, then other transactions have no way of telling whether potential readers of an old version still exist! To circumvent this apparent paradox, we exploit separate GC threads, such as those available in managed memory systems. Such threads have access to all the threads' private memories, so that even operations that are invisible to other transactions are visible to the garbage collector. SMV ensures that old object versions become *garbage collectible* once there are no transactions that can safely read them.

In Section 5 we evaluate different aspects of SMV's performance. We implement SMV in Java and study its behavior for a number of benchmarks (red-black tree microbenchmark, STMBench7 [18] and Vacation [8]).

We find that SMV is extremely efficient for read-dominated workloads with long-running transactions. For example, in STMBench7 with 64 threads, the throughput of SMV is seven times higher than that of TL2 and more than double than those of 2- and 8-versioned STMs. Furthermore, in an application with one thread constantly taking snapshots and the others running update transactions, neither TL2 nor the k-versioned STM succeeds in taking a snapshot, even when only one concurrent updater is running. The performance of SMV remains stable for any number of concurrent updaters.

We compare the memory demands of the algorithms by limiting Java heap size. Whereas k-versioned STMs crash with a Java `OutOfMemoryException`, SMV continues to run, and its throughput is degraded by less than 25% even under stringent memory constraints.

In summary, we present the new approach for keeping multiple versions, which allows read-only transactions to stay invisible and delegates the cleanup task to the already existing GC mechanisms. Our conclusions appear in Section 6.

2 Related Work

As noted above, most existing STMs are single-versioned. Of these, SMV is most closely related to TL2 [11], from which we borrow the ideas of invisible reads, commit-time locking of updated objects, and a global version clock for consistency checking. In a way, SMV can be seen as a multi-versioned extension of TL2.

The best-known representative of multi-versioned STMs is LSA [29]. LSA, as well as its snapshot-isolation variation [30], implements a simple solution to garbage collection: it keeps a constant number of versions for each object. However, this approach leads to storing versions that are too old to be of use to any transaction on the one hand, and to aborting transactions because they need older versions than the ones stored on the other. In contrast, SMV keeps versions as long as they might be useful for ongoing transactions, and makes them GCable by an automatic garbage collector as soon as they are not. For infrequently updated objects, SMV typically keeps a single version.

Another multi-versioned STM, JVSTM [7], maintains a priority queue of all active transactions, sorted by their start time. A cleanup thread waits until the transaction at the head of the queue (the oldest transaction) is finished. When that happens, the cleanup process iterates over the objects overwritten by the committed transaction and discards their previous versions. Thus, while also keeping versions only as long as active transactions might read them, the GC mechanism of JVSTM imposes an additional overhead for transaction startup and termination (including both update and read-only transactions).

In a recent paper [15], the authors improved the GC mechanism of JVSTM by maintaining a global list of per-thread transactional contexts, each keeping information about the latest needed versions. A special cleanup thread iterates periodically over this list and thus finds the versions that can be discarded. This improvement, however, does not eliminate the need for a special cleanup thread, which should run in addition to Java GC threads. JVSTM read-only transactions still need to write to the global memory. In contrast, in this paper we present a simple algorithm with invisible read-only transactions, which exploits the automatic GC available in languages with managed memory.

Other previous suggestions for multi-versioned STMs [3,25,22,6,27] were based on cycle detection in the conflict graph, a data structure representing all data dependencies among transactions. This approach incurs a high cost (quadratic in the number of transactions), which is clearly not practical. Moreover, it requires reads to be visible in order to detect future conflicts, which can be detrimental to performance. Nevertheless, this approach can allow for more accurate garbage collection than practical systems like SMV, which do not maintain conflict graphs. For example, our earlier work [22,27] specified GC rules based on precedence information as to when old versions can be removed. However, in addition to being too complex to be amenable to practical implementation, these earlier works did not specify when these GC rules ought to be checked. Aydonat and Abdelrahman [3] propose to keep each version for as long as transactions that were active at the time the version was created exist, but the authors do not specify how this rule can be implemented efficiently. Other theoretical suggestions for multi-versioned STMs ignored the issue of GC altogether [25].

Instead of multi-versioning, STMs can avoid aborts by reading uncommitted values and then having the reader block until the writer commits [28], or by using read-write locks to block in case of concurrency [12,2]. These approaches differ from SMV, where transactions never block and may always progress independently. Moreover, reads, which are invisible in SMV, must be visible in these "blocking" approaches. In addition, reading the values of uncommitted transactions might lead to cascading aborts.

Transactional mutex locks (TML) [10], have been shown to be very efficient for read-dominated workloads due to their simplicity and low overhead. Unlike SMV, TML do not allow concurrency between update transactions and thus do not exploit the parallelism in read-write or write-dominated workloads.

Another technique for reducing the number of aborts is timestamp extension [29,14]. This mechanism requires maintaining a read-set and therefore is usually not used by read-only transactions. Timestamp extension is applicable for SMV's update transactions as well, hence this improvement is orthogonal to the multi-versioning approach presented in this paper.

3 Exponential Memory Growth

Before introducing SMV, we first describe an inherent memory consumption problem of algorithms keeping a constant number of object versions. A naïve assessment of the memory consumption of a k-versioned STM would probably estimate that it takes up to k times as much more memory as a single-versioned STM.

We now illustrate that, in fact, the memory consumption of a k-versioned STM in runs with n transactional objects might grow like k^n. Intuitively, this happens because previous object versions continue to keep references to already deleted objects, which causes deleted objects to be pinned in memory.

Consider, for example, a 2-versioned STM in the scenario depicted in Figure 1. The STM keeps a linked list of three nodes. When removing node 30 and inserting a new node 40 instead, node 30 is still kept as the previous version of 20.*next*. Next, when node 20 is replaced with node 25, node 30 is still pinned in memory, as it is referenced by node 20. After several additional node replacements, we see that there is a complete binary tree in memory, although only a linked list is used in the application.

More generally, with a k-versioned STM, a linked list of length n could lead to $\Omega(k^n)$ node versions being pinned in memory (though being still linear to the number of write

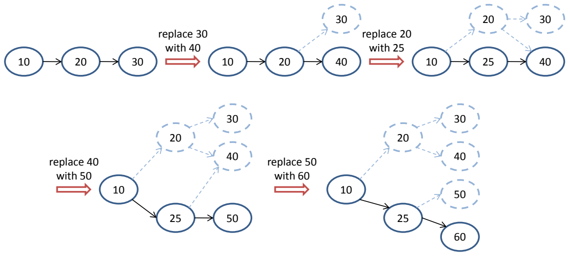


Fig. 1. Example demonstrating exponential memory growth even for an STM keeping only 2 versions of each object. A linked list causes a binary tree to be pinned in memory because previous node versions continue to keep references to already deleted nodes.

operations). This demonstrates an inherent limitation of keeping a constant number of versions per object. Our observation is confirmed by the empirical results shown in Section 5.5, where the algorithms keeping k versions cannot terminate in the runs with a limited heap size, while SMV does not suffer from any serious performance degradation.

4 SMV Algorithm

We present Selective Multi-Versioning, a new object-based STM. The data structures used by SMV are described in Section 4.1 and Section 4.2 depicts the algorithm.

4.1 Overview of Data Structures

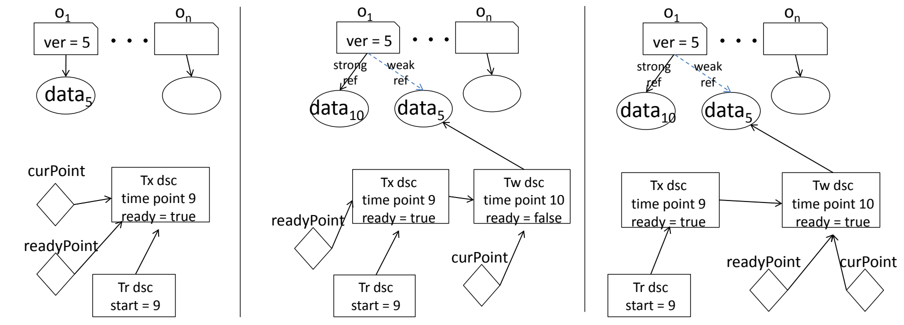
SMV's main goal is to reduce aborts in workloads with read-only transactions, without introducing high space or computational overheads. SMV is based on the following design choices: 1) Read-only transactions do not affect the memory that can be accessed by other transactions. This property is important for performance in multi-core systems, as it avoids cache thrashing issues [13,29]. 2) Read-only transactions always commit. A read-only transaction T_i observes a consistent snapshot corresponding to T_i's start time — when T_i reads object o_j, it finds the latest version of o_j that has been written before T_i's start. 3) Old object versions are removed once there are no live read-only transactions that can consistently read them. To achieve this with invisible reads, SMV relies on the omniscient GC mechanism available in managed memory systems.

We now give a brief reminder of such a mechanism. An object can be reclaimed by the garbage collector once it becomes unreachable from the call stack or global variables. Reachability is a transitive closure over *strong* memory references: if a reachable object o_1 has a strong reference to o_2, then o_2 is reachable as well (strong references are the default ones in Java). In contrast, *weak references* [17] do not protect the referenced object from being GCed; an object referenced by weak references only is considered unreachable and may be removed.

As in other object-based STMs, transactional objects in SMV are accessed via *object handles*. An object handle includes a history of object values, where each value keeps a *versioned lock* [11] – data structure with a version number and a lock bit. In order to facilitate automatic garbage collection, object handles in SMV keep strong references only to the latest (current) versions of each object, and use weak references to point to other versions.

Each transaction is associated with a *transactional descriptor*, which holds the relevant transactional data, including a read-set, a write-set, status, etc. In addition, transactional descriptors play an important role in keeping strong references to old object versions, as we explain below.

Version numbers are generated using a global version clock, where transactional descriptors act as "time points" organized in a one-directional linked list. Upon commit, an update transaction appends its transactional descriptor to the end of the list (a special global variable *curPoint* points to the latest descriptor in this list). For example, if the current global version is 100, a committing update transaction sets the time point value in its transactional descriptor to 101 and adds a pointer to this descriptor from the descriptor holding 100.



(a) T_r's descriptor points to the latest committed transaction.
(b) T_w commits and begins write-back.
(c) T_w's write-back is finished.

Fig. 2. Transactional descriptor of T_w references the over-written version of o_1 (data$_5$). This way, read-only transaction T_r keeps a reference chain to the versions that have been overwritten after T_r's start.

Version management is based on the idea that old object versions are pointed to by the descriptors of transactions that over-wrote these versions (see Figure 2). A committing transaction T_w includes in its transactional descriptor a strong reference to the previous version of every object in its write set before diverting the respective object handle to the new version.

When a read-only transaction T_i begins, it keeps (in its local variable *startTP*) a pointer to the latest transactional descriptor in the list of committed transactions. This pointer is cleared upon commit, making old transactional descriptors at the head of the list GCable.

This way, active read-only transaction T_r keeps a reference chain to version o_i^j if this version was over-written after T_r's start, thus preventing o_i^j's garbage collection. Once there are no active read-only transactions that started before o_i^j was over-written, this version stops being referenced and thus becomes GCable .

Figure 2 illustrates the commit of an update transaction T_w that writes to object o_1 (the use of *readyPoint* variable will be explained in Section 4.3). In this example, T_w and a read-only transaction T_r both start at time 9, and hence T_r references the transactional descriptor of time point 9. The previous update of o_1 was associated with version 5. When T_w commits, it inserts its transactional descriptor at the end of the time points list with value 10. T_w's descriptor references the previous value of o_1. This way, the algorithm creates a reference chain from T_r to the previous version of o_1 via T_w's descriptor, which ensures that the needed version will not be GCed as long as T_r is active.

4.2 Basic Algorithm

We now describe the SMV algorithm. For the sake of simplicity, we present the algorithm in this section using a global lock for treating concurrency on commit — in Section 4.3 we show how to remove this lock.

SMV handles read-only and update transactions differently. We assume that transaction's type can be provided to the algorithm beforehand by a compiler or via special

Algorithm 1. SMV algorithm for update transaction T_i.

```
 1: Upon Startup:
 2:     $T_i$.startTime ← curPoint.commitTime

 3: Read $o_j$:
 4:     if ($o_j \in T_i$.writeSet)
 5:         then return $T_i$.writeSet[$o_j$]
 6:     data ← $o_j$.latest
 7:     if ¬validateRead($o_j$) then abort
 8:     readSet.put($o_j$)
 9:     return data

10: Write to $o_j$:
11:     if ($o_j \in T_i$.writeSet)
12:         then update $T_i$.writeSet.get($o_j$); return
13:     localCopy ← $o_j$.latest.clone()
14:     update localCopy; writeSet[$o_j$] ← localCopy

15: Function validateReadSet
16:     foreach $o_j \in T_i$.readSet do:
17:         if ¬validateRead($o_j$) then return false
18:     return true

19: Commit:
20:     foreach $o_j \in T_i$.writeSet do: $o_j$.lock()
21:     if ¬validateReadSet() then abort
        ▷ txn dsc should reference the over-written data
22:     foreach $o_j \in T_i$.writeSet do:
23:         $T_i$.prevVersions.put(⟨$o_j$, $o_j$.latest⟩)
24:     timeLock.lock()
25:     $T_i$.commitTime ← curPoint.commitTime + 1
        ▷ update and unlock the objects
26:     foreach ⟨$o_j$, data⟩ $\in T_i$.writeSet do:
27:         $o_j$.version ← $T_i$.commitTime
28:         $o_j$.weak_references.append($o_j$.latest)
29:         $o_j$.latest ← data; $o_j$.unlock()
30:     curPoint.next ← $T_i$; curPoint ← $T_i$
31:     timeLock.unlock()

32: Function validateRead(Object $o_j$)
33:     return (¬$o_j$.isLocked ∧ $o_j$.version ≤ $T_i$.startTime)
```

program annotations. If not, each transaction can be started as read-only and then restarted as update upon the first occurrence of a write operation.

Handling update transactions. The protocol for update transaction T_i is depicted in Algorithm 1. The general idea is similar to the one used in TL2 [11]. An update transaction T_i aborts if some object o_j read by T_i is over-written after T_i begins and before T_i commits. Upon starting, T_i saves the value of the latest time point in a local variable *startTime*, which holds the latest time at which an object in T_i's read-set is allowed to be over-written.

A read operation of object o_j reads the latest value of o_j, and then post-validates its version (function *validateRead*. The validation procedure checks that the version is not locked and it is not greater than T_i.startTime, otherwise the transaction is aborted.

A write operation (lines 12–14) creates a copy of the object's latest version and adds it to T_i's local write set.

Commit (lines 20–31) consists of the following steps:

1. Lock the objects in the write set (line 20). Deadlocks can be detected using standard mechanisms (e.g., timeouts or Dreadlocks [24]), or may be avoided if acquired in the same order by every transaction.
2. Validate the read set (function *validateReadSet*).
3. Insert strong references to the over-written versions to T_i's descriptor (line 23). This way the algorithm guarantees that the over-written versions stay in the memory as long as T_i's descriptor is referenced by some read-only transaction.
4. Lock the time points list (line 24). Recall that this is a simplification; in Section 4.3 we show how to avoid such locking.
5. Set the commit time of T_i to one plus the value of the commit time of the descriptor referenced by *curPoint*.
6. Update and unlock the objects in the write set (lines 26–29). Set their new version numbers to the value of T_i.commitTime. Keep weak references to old versions.
7. Insert T_i's descriptor to the end of the time points list and unlock the list (line 30).

Algorithm 2. SMV algorithm for read-only transaction T_i.

```
1: Upon Startup:
2:     $T_i$.startTP ← curPoint

3: Read $o_j$:
4:     latestData ← $o_j$.latest
5:     if ($o_j$.version $\leq$ $T_i$.startTP.commitTime) then return latestData
6:     return the latest version ver in $o_j$.weak_references, s.t.
7:         ver.version $\leq$ $T_i$.startTP.commitTime

8: Commit:
9:     $T_i$.startTP ← $\perp$
```

Handling read-only transactions. The pseudo-code for read-only transactions appears in Algorithm 2. Such transactions always commit without waiting for other transactions to invoke any operations. The general idea is to construct a consistent snapshot based on the start time of T_i. At startup, T_i*.startTP* points to the latest installed transactional descriptor (line 2); we refer to the time value of startTP as T_i*'s start time*.

For each object o_j, T_i reads the latest version of o_j written before T_i's start time. When T_i reads an object o_j whose latest version is greater than its start time, it continues to read older versions until it finds one with a version number older than its start time. Some old enough version is guaranteed to be found, because the updating transaction T_w that over-wrote o_j has added T_w's descriptor referencing the over-written version somewhere after T_i's starting point, preventing GC.

The commit procedure for read-only transactions merely removes the pointer to the starting time point, in order to make it GCable, and always commits.

4.3 Allowing Concurrent Access to the Time Points List

We show now how to avoid locking the time points list (lines 24, 31 in Algorithm 1), so that update transactions with disjoint write-sets may commit concurrently.

We first explain the reason for using the lock. In order to update the objects in the write-set, the updating transaction has to know the new version number to use. However, if a transaction exposes its descriptor before it finishes updating the write-set, then some read-only transaction might observe an inconsistent state. Consider, for example, transaction T_w that updates objects o_1 and o_2. The value of *curPoint* at the beginning of T_w's commit is 9. Assume T_w first inserts its descriptor with value 10 to the list, then updates object o_1 and pauses. At this point, $o_1.version = 10$, $o_2.version < 10$ and $curPoint \rightarrow commitTime = 10$. If a new read-only transaction starts with time 10, it can successfully read the new value of o_1 and the old value of o_2, because they are both less than or equal to 10. Intuitively, the problem is that the new time point becomes available to the readers as a potential starting time before all the objects of the committing transaction are updated.

To preserve consistency without locking the time points list, we add an additional boolean field *ready* to the descriptor's structure, which becomes *true* only after the committing transaction finishes updating all objects in its write-set. In addition to the global *curPoint* variable referencing the latest time point, we keep a global *readyPoint* variable, which references the latest time point in the *ready prefix* of the list (see Figure 2).

When a new read-only transaction starts, its *startTP* variable references *readyPoint*. In the example above, a new transaction T_r begins with a start time equal to 9, because the new time point with value 10 is still not ready. Generally, the use of *readyPoint* guarantees that if a transaction reads an object version written by T_w, then T_w and all its preceding transactions had finished writing their write-sets.

Note, however, that when using ready points we should not violate the real time order — if a read-only transaction T_r starts after T_w terminates, then T_r must have a start time value not less than T_w's commit time. This property might be violated if update transactions become ready in an order that differs from their time points order, thus leaving an unready transaction between ready ones in the list.

We have implemented two approaches to enforce real-time order: 1) An update transaction does not terminate until the ready point reaches its descriptor. A similar approach was previously used by RingSTM [33] and JVSTM [15]. 2) A new read-only transaction notes the time point of the latest terminated transaction and then waits until the *readyPoint* reaches this point before starting. Note that unlike the first alternative, read-only transactions in the second approach are not wait-free.

According to our evaluation, both techniques demonstrate similar results. The waiting period remains negligible as long as the number of transactional threads does not exceed the number of available cores; when the number of threads is two times the number of cores, waiting causes a $10 - 15\%$ throughput degradation (depending on the workload) — this is the cost we pay for maintaining real-time order.

5 Implementation and Evaluation

5.1 Compared Algorithms

Our evaluation aims to check the aspect of keeping and garbage collecting multiple versions. Direct comparison was difficult because of different frameworks the algorithms are implemented in[1]. We implement the following algorithms:

- **SMV**– The algorithm described in Section 4.
- **TL2**– Java implementation of TL2 [11] with a single central global version clock. We use a standard optimization of not keeping a read-set for read-only transactions. The code follows the style of TL2 implementation in Deuce framework [23].
- **TL2 with time points**– A variant of TL2, which implements the time points mechanism described in Section 4.1. This way, we check the influence of the use of time points on overall performance and separate it from the impact of multi-versioning techniques used in SMV.
- k**-versioned**– an STM based on a TL2-style's logic and code, in which each object keeps a constant k, number of versions (this approach resembles LSA [29]). Reads operate as in SMV, except that if no adequate version is found, the transaction aborts. Updates operate as in TL2.
- **Read-Write lock (RWLock)**– a simple global read-write lock. The lock is acquired at the beginning of an atomic section and is released at its end.

[1] DeuceSTM [23] framework comes with TL2 and LSA built-in, however, its LSA implementation is single-versioned.

We use the Polite contention manager with exponential backoff [31] for all the algorithms: aborted transactions spin for a period of time proportional to 2^n, where n is the number of retries of the transaction.

5.2 Experiment Setup

All algorithms are implemented in Java. We use the following benchmarks for performance evaluation: 1) a red-black tree microbenchmark; 2) the Java version of STMBench7 [18]; and 3) Vacation, which is part of the STAMP [8] benchmark suite.

Red-black tree microbenchmark. The red-black tree supports insertion, deletion, query and range query operations. The initial size of the tree is 400000 nodes. It is checked both for read-dominated workloads ($80/20$ ratio of read-only to update operations) and for workloads with update operations only.

STMBench7. STMBench7 aims to simulate different behaviors of real-world programs by invoking both read-only and update transactions of different lengths over large data structures, typically graphs. Workload types differ in their ratio of read-only to update transactions: $90/10$ for *read-dominated* workloads, $60/40$ for *read-write* workloads, and $10/90$ for *write-dominated* workloads. When running STMBench7 workloads, we bound the length of each benchmark by the number of transactions performed by each thread (2000 transactions per thread unless stated otherwise). We manually disabled long update traversals because they inherently eliminate any potential for scalability.

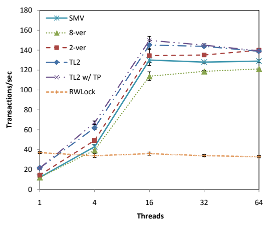
Vacation (Java port). Vacation [8], emulates a travel reservation system, which is implemented as a set of trees. In our experiments it is run with the standard parameters corresponding to `vacation-high++`. Note that STAMP benchmarks are not suitable for evaluating techniques that optimize read-only transactions, because these benchmarks do not have read-only transactions at all. We use Vacation as one exemplary STAMP application to evaluate SMV's overhead.

Setup. The benchmarks are run on a dedicated shared-memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor. The system runs Linux 2.6.22.5-31 with swap turned off. For all tests but those with limited memory, JVM is run with the `AggressiveHeap` flag on. Thread scheduling is left entirely to the OS. We run up to 64 threads on the 32 cores.

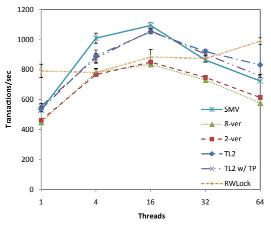
Our evaluation study is organized as follows: in Section 5.3, we show system performance measurements. Section 5.4 considers the latency and predictability of long read-only operations, and in Section 5.5, we analyze the memory demands of the algorithms.

5.3 Performance Measurements

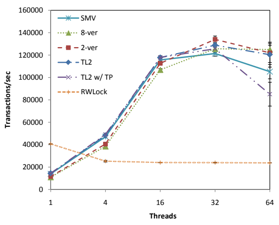
SMV overhead. As we mentioned earlier, the use of multiple versions in our algorithm can be exploited by read-only transactions only. However, before evaluating the performance of SMV with read-only transactions, we first want to understand its behavior in programs with update transactions only. In these programs, SMV can hardly be expected to outperform its single-versioned counterparts. For update transactions, SMV resembles the behavior of TL2, with the additional overhead of maintaining previous



(a) Throughput in red-black tree write-only workload

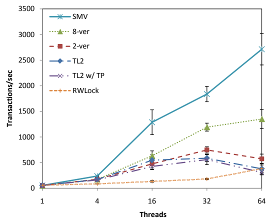


(b) Throughput in STMBench7's write-dominated workload

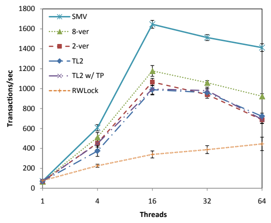


(c) Throughput in Vacation benchmark

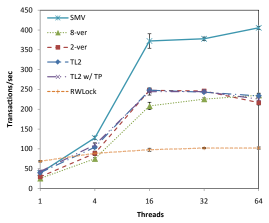
Fig. 3. In the absence of read-only transactions multi-versioning cannot be exploited. The overhead of SMV degrades throughput by up to 15%.



(a) Throughput in STMBench7's read-dominated workload



(b) Throughput in STMBench7's read-write workload



(c) Throughput in the RBTree read-dominated workload

Fig. 4. By reducing aborts of read-only transactions, SMV presents a substantially higher throughput than TL2 and the k-versioned STM. In read-dominated workloads, its throughput is $\times 7$ higher than that of TL2 and more than twice those of the k-versioned STM with $k = 2$ or $k = 8$. In read-write workloads its advantage decreases because of update transactions, but SMV still clearly outperforms its competitors.

object versions. Thus, measuring throughput in programs without read-only transactions quantifies the cost of this additional overhead.

In Figure 3, we show throughput measurements for write-dominated benchmarks: Red-black tree (Figure 3(a)) and Vacation (Figure 3(c)) do not contain read-only transactions at all. The write-dominated STMBench7 workload shown in Figure 3(b) runs 90% of its operations as update transactions, and therefore the influence of read-only ones is negligible.

All compared STM algorithms show similar behavior in all three benchmarks. This emphasizes the fact that the algorithms take the same approach when executing update transactions and that they all have a common underlying code platform. The differences in the behavior of RWLock are explained by different contention levels of the benchmarks. While the contention level in Vacation remains moderate even for 64 threads, contention in the write-dominated STMBench7 is extremely high, so that RWLock outperforms the other alternatives.

Figure 3 demonstrates low overhead of SMV when the number of threads does not exceed 32; for 64 threads this overhead causes a 15% throughput drop. This is the cost we pay for maintaining multiple versions when these versions are not actually used.

Throughput. We next run workloads that include read-only transactions, in order to assess whether the overhead of SMV is offset by its smaller abort rate. In Figure 4 we depict throughput measurements of the algorithms in STMBench7's read-dominated and read-write workloads, as well as the throughput of the red-black tree. We see that in the read-dominated STMBench7 workload, SMV's throughput is seven times higher than that of TL2. Despite keeping as many as 8 versions, the k-versioned STM cannot keep up, and SMV outperforms it by more than twice.

What is the reason for 8 versions not being enough? In the full version of the paper [26] we show the following two results that explain this: First, the probability of accessing an old object version is extremely small (less than 2.5% even for the second version). Therefore, keeping k versions for *each object* can be wasteful. Second, the amount of work lost because the k^{th} version is absent, is surprisingly high even for large k values. Intuitively, this occurs since a transaction that needs to access the k^{th} version of an object must have been running for a long time, and the price of aborting such a transaction is high. Hence, keeping previous versions is important despite the low frequency of accessing them; keeping a constant number of versions per object will typically not be enough for reducing the amount of wasted work.

We further note that SMV is scalable, and its advantage over a single-version STM becomes more pronounced as the number of threads rises. In the read-write workload, the number of read-only transactions that can use multiple versions decreases, and the throughput gain becomes 95% over TL2 and 52% over the 8-versioned STM.

We conclude that in the presence of read-only transactions the benefit of SMV significantly outweighs its overhead. In the full version of the paper [26] we explain this benefit by looking at the amount of work wasted due to aborts. We show that in the read-dominated workload, TL2 spends more than 80% of its time running aborted transactions! Interestingly, k-versioned STMs cannot fully alleviate this effect, reducing the amount of wasted time only to 36%. In contrast, SMV's waste does not rise above 3%.

It is possible to employ timestamp extension [29,14] to reduce the amount of wasted work in both TL2 and SMV. However, this approach requires read-only transactions to maintain read-sets. The overhead of keeping a read-set is significant for long read-only transactions. We implemented timestamp extension in both TL2 and SMV, and our experiments showed that it did not improve the performance of either algorithm, although it did reduce the amount of wasted work. For space imitations, we omit these results.

5.4 Latency and Predictability of Long Read-Only Operations

In the previous section we concentrated on overall system performance without considering specific transactions. However, in real-life applications the completion time of individual operations is important as well. In this section we consider two examples: taking system snapshots of a running application and STMBench7's long traversals.

Taking a full-system snapshot is important in various fields: it is used in client-server finance applications to provide clients with consistent views of the state, for checkpointing in high-performance computing, for creating new replicas, for application monitoring and gathering statistics, etc. Predictability of the time it takes to complete the snapshot is important, both for program stability and for usability.

Table 1. Maximum time for completing long read-only operations. Long read-only traversals in STMBench7 can be hardly predictable for TL2 and k-versioned STMs: they might take hundreds of seconds under high loads. Vacation snapshot operation run by TL2 or k-versioned algorithms cannot terminate even when there is only a single application thread. SMV presents stable performance unaffected by the level of contention both for STMBench7 traversals and Vacation snapshots.

(a) Maximum time (sec) for completing a long read-only operation in STMBench7.

	Number of threads				
	1	4	8	16	32
TL2	1.3	21.6	68.5	103.6	358.5
SMV	1.3	1.4	2.4	3.6	11.9
2-versioned	1.3	4.1	22.9	45.2	204.5
8-versioned	1.3	6.8	10.6	22.2	79.4

(b) Maximum time (sec) to take a snapshot in Vacation benchmark.

	Number of threads				
	1	4	8	16	32
TL2	—	—	—	—	—
SMV	1.4	1.3	1.2	1.4	1.5
2-versioned	—	—	—	—	—
8-versioned	—	—	—	—	—

We first show the maximum time for completing a long read-only traversal, which is already built-in in STMBench7 (see Table 1(a)). As we can see from the table, this operation takes only several seconds when run without contention. However, when the number of threads increases, completing the traversal might take more than 100 seconds in TL2 and k-versioned STMs. Unlike those algorithms, SMV is less impacted by the level of contention and it always succeeds to complete the traversal in several seconds.

Next, we added the option of taking a system snapshot in Vacation. In addition to the original application threads, we run a special thread that repeatedly tries to take a snapshot. We are interested in the maximum time it takes to complete the snapshot operation. The results appear in Table 1(b). We see that neither TL2 nor the k-versioned STM can successfully take a snapshot even when only a single application thread runs updates in parallel with the snapshot operation. Surprisingly, even 8 versions do not suffice to allow snapshots to complete, this is because within the one and a half seconds it takes the snapshot to complete some objects are overwritten more than 8 times.

On the other hand, the performance of SMV remains stable and unaffected by the number of application threads in the system. We conclude that SMV successfully keeps the needed versions. In Section 5.5, we show that it does so with smaller memory requirements than the k-versioned STM.

We would like to note that while taking a snapshot is also possible by pausing mutator threads, this approach is much less efficient, as it requires quiescence periods and thus reduces the overall throughput.

5.5 Memory Demands

One of the potential issues with multi-versioned STMs is their high memory consumption. In this section we compare memory demands of the different algorithms. To this end, we execute long-running write-dominated STMBench7 benchmarks (64 threads, each thread running 40000 operations) with different limitations on the Java memory heap. Such runs present a challenge for the multi-versioned STMs because of their high update rate and limited memory resources. As we recall from Section 5.3, multi-versioned STMs cannot outperform TL2 in a write-dominated workload. Hence, the goal of the current experiment is to study the impact of the limited memory availability on the algorithms' behaviors.

Figure 5 shows how the algorithms' throughput depends on the Java heap size. A "—" sign corresponds to runs in which the algorithm did not succeed to complete the benchmark due to a Java `OutOfMemoryException`. Notice that the 8-versioned STM is unable to successfully complete a run even given a 16GB Java heap size. Decreasing k to 4, and then 2, makes it possible to finish the runs under stricter constraints. However, none of the k-versioned STMs succeed under the limitation of 2GB. Unlike k-versioned STMs, SMV continues to function under these constraints. Furthermore, SMV's throughput does not change drastically — the maximum decrease is 25% when Java heap size shrinks 8-fold.

	Memory limit				
	2GB	4GB	8GB	12GB	16GB
TL2	606.89	631.56	630.3	674.96	647.17
SMV	450.12	543.04	563.74	595.78	602.01
2-versioned	—	515.32	532.7	550.61	533.01
4-versioned	—	—	—	—	281.98
8-versioned	—	—	—	—	—

Fig. 5. Throughput (txn/sec) in limited memory systems: k-versioned STMs do not succeed to complete the benchmark

The collapse of the k-versioned STM confirms the observation from Section 3, where we have illustrated that its memory consumption can become exponential rather than linear in the number of transactional objects.

6 Conclusions

Many real-world applications invoke a high rate of read-only transactions, including ones executing long traversals or obtaining atomic snapshots. For such workloads, multi-versioning is essential: it bears the promise of high performance, reduced abort rates, and less wasted work.

We presented Selective Multi-Versioning, a new STM that achieves high performance (high throughput, low and predictable latency, and little wasted work) in the presence of long read-only transactions. Despite keeping multiple versions, SMV can work well in memory constrained environments.

SMV keeps old object versions as long as they might be useful for some transaction to read. We do so while allowing read-only transactions to remain invisible by relying on automatic garbage collection to dispose of obsolete versions.

References

1. `http://www.azulsystems.com/blog/cliff-click/2008-05-27-clojure-stms-vs-locks`
2. Attiya, H., Hillel, E.: Brief announcement: Single-Version STMs can be Multi-Version Permissive. In: Proceedings of the 29th Symposium on Principles of Distributed Computing (2010)
3. Aydonat, U., Abdelrahman, T.: Serializability of transactions in software transactional memory. In: Second ACM SIGPLAN Workshop on Transactional Computing (2008)
4. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, pp. 1–10 (1995)

5. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
6. Bieniusa, A., Fuhrmann, T.: Consistency in hindsight, a fully decentralized stm algorithm. In: IPDPS 2010: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (2010)
7. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming 63(2), 172–185 (2006)
8. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC 2008: Proceedings of The IEEE International Symposium on Workload Characterization (September 2008)
9. Carvalho, N., Cachopo, J., Rodrigues, L., Rito-Silva, A.: Versioned transactional shared memory for the FenixEDU web application. In: Proceedings of the 2nd Workshop on Dependable Distributed Data Management, pp. 15–18 (2008)
10. Dalessandro, L., Dice, D., Scott, M., Shavit, N., Spear, M.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 2–13. Springer, Heidelberg (2010)
11. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
12. Dice, D., Shavit, N.: TLRW: Return of the read-write lock. In: TRANSACT 2009: 4th Workshop on Transactional Computing (February 2009)
13. Ennals, R.: Cache sensitive software transactional memory. Technical report
14. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008, pp. 237–246 (2008)
15. Fernandes, S.M., Cachopo, J.A.: Lock-free and Scalable Multi-Version Software Transactional Memory. In: PPoPP 2011, pp. 179–188 (2011)
16. Fraser, K.: Practical lock freedom. PhD thesis. Cambridge University Computer Laboratory (2003)
17. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
18. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A Benchmark for Software Transactional Memory. In: Proceedings of the Second European Systems Conference (2007)
19. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003, pp. 92–101 (2003)
20. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
21. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
22. Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. In: SPAA 2009, pp. 59–68 (2009)
23. Korland, G., Shavit, N., Felber, P.: Noninvasive Java concurrency with Deuce STM (poster). In: SYSTOR 2009 (2009), Further details at http://www.deucestm.org/
24. Koskinen, E., Herlihy, M.: Dreadlocks: efficient deadlock detection. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 297–303 (2008)
25. Napper, J., Alvisi, L.: Lock-free serializable transactions. Technical report, The University of Texas at Austin (2005)
26. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective Multi-Versioning STM. Technical report, Technion (2011)
27. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in transactional memory. In: PODC (2010)

28. Ramadan, H.E., Roy, I., Herlihy, M., Witchel, E.: Committing conflicting transactions in an STM. SIGPLAN Not. 44(4), 163–172 (2009)
29. 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)
30. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: 1st ACM SIGPLAN Workshop on Transactional Computing, TRANSACT (2006)
31. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC 2005, pp. 240–248 (2005)
32. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 204–213 (1995)
33. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: scalable transactions with a single atomic instruction. In: SPAA 2008, pp. 275–284 (2008)

Brief Announcement: Leaderless Byzantine Paxos

Leslie Lamport

Microsoft Research

Abstract. The role of leader in an asynchronous Byzantine agreement algorithm is played by a virtual leader that is implemented using a synchronous Byzantine agreement algorithm.

Agreement in a completely asynchronous distributed system is impossible in the presence of even a single fault [5]. Practical fault-tolerant "asynchronous" agreement algorithms assume some synchrony assumption to make progress, maintaining consistency even it that assumption is violated. Dependence on synchrony may be explicit [4], or may be built into reliance on a failure detector [2] or a leader-election algorithm. Algorithms that are based on leader election are called Paxos algorithms [6,7,8]. Byzantine Paxos algorithms are extensions of these algorithms to tolerate malicious failures [1,9].

For Byzantine agreement, reliance on a leader is problematic. Existing algorithms have quite convincing proofs that a malicious leader cannot cause inconsistency. However, arguments that a malicious leader cannot prevent progress are not so satisfactory. Castro and Liskov [1] describe a method by which the system can detect lack of progress and choose a new leader. However, their method is rather ad hoc. It is not clear how well it will work in practice, where it can be very difficult to distinguish malicious behavior from transient communication errors.

The first Byzantine agreement algorithms, developed for process control applications, did not require a leader [10]. However, they assumed synchronous communication: that messages sent between nonfaulty processes are received within a known length of time. These algorithms are not suitable in the asynchronous case because a loss of synchrony can cause inconsistency.

We propose a simple method for implementing a leaderless Byzantine agreement algorithm: replacing the leaders in an ordinary Byzantine Paxos algorithm by a virtual leader that is implemented using a synchronous Byzantine agreement algorithm. Messages that in the ordinary algorithm are sent to the leader are instead sent to all the servers. Each server then decides what message the leader should send next and proposes it as the leader's next message. The servers then execute a synchronous Byzantine agreement algorithm to try to agree on the vector of proposed messages—a vector containing one proposal for each server. (This type of agreement is called *interactive consistency* [10].) Each server then uses a deterministic procedure to choose the message sent by the virtual leader, and it acts as if it had received this message.

When the system behaves synchronously, as is required for progress by any algorithm, each non-faulty server chooses the same virtual-leader message. The virtual leader thus behaves correctly, and the Byzantine Paxos algorithm makes progress. If the system does not behave synchronously, then the synchronous Byzantine agreement algorithm may fail, causing different servers to choose different virtual-leader messages. This is equivalent to a malicious leader sending conflicting messages to different processes. The malicious virtual leader can prevent progress (which cannot be guaranteed without synchrony), but does not cause inconsistency because a Byzantine Paxos algorithm can tolerate a malicious leader.

Leaderless Paxos adds to a Byzantine Paxos algorithm the cost of the leader agreement algorithm. The time required by a leader agreement algorithm that tolerates F faulty servers is $F+1$ message delays, which replaces the 1 message delay of a leader simply sending a message. (Early-stopping algorithms probably cannot be used because implementing a virtual leader seems to require simultaneous Byzantine agreement, which cannot guarantee early stopping [3].) For N servers, approximately NF extra messages are required.

References

1. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation, pp. 173–186. ACM, New York (1999)
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
3. Dolev, D., Reischuk, R., Strong, H.R.: Early stopping in Byzantine agreement. Journal of the ACM 37(4), 720–741 (1990)
4. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
5. Fischer, M.J., Lynch, N., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
6. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
7. Lamport, L.: Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32(4), 51–58 (2001)
8. Lamport, L.: Fast paxos. Distributed Computing 19(2), 79–103 (2006)
9. Martin, J.-P., Alvisi, L.: Fast byzantine consensus. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005), Yokohama, pp. 402–411. IEEE Computer Society, Los Alamitos (2006)
10. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27(2), 228–234 (1980)

Brief Announcement: When You Don't Trust Clients: Byzantine Proposer Fast Paxos

Keith Marzullo[1], Hein Meling[2], and Alessandro Mei[3,⋆]

[1] Dept. of Computer Science and Engineering, University of California, San Diego
[2] Dept. of Computer Science and Engineering, University of Stavanger, Norway
[3] Dept. of Computer Science, Sapienza University of Rome, Italy

1 Introduction

State machine replication is a general approach for constructing fault-tolerant services, and a key protocol underlying state machine replication is consensus. The set of Byzantine failures is so large that it has been applied for masking the effects of compromised systems, and so Byzantine-tolerant consensus has been used to construct systems that are meant to ameliorate the effect of compromise (see [1] among others). In the Byzantine model, there is no trust among processes: any process can behave in an arbitrarily faulty manner. However, in multi-site systems, processes in the same administrative domain typically have a measure of mutual trust. This is because such processes share fate: for example, if a process in a domain is compromised, then other processes—perhaps all of them—can be compromised as well, and the local services they rely upon may be compromised. In [4], this observation was used to argue for the *Mutually Suspicious Domain* (MSD) model, in which there is mutual trust between processes in a domain, but no trust for inter-domain communication, i.e., processes within a domain must protect itself from possible uncivil behavior from processes in other domains.

In this paper, we propose a consensus protocol for state machine replication under the MSD model. We assume the typical Internet model, in which the servers are in the same administrative domain and replicated for increased availability, and clients are in other administrative domains. The protocol, which we call BP Fast Paxos, uses a hybrid failure model: processes within a domain assume a crash failure model while across domains they assume a Byzantine failure model. BP Fast Paxos is derived from Paxos [2], and provides low latency for client requests, can tolerate any number of (Byzantine) faulty clients, up to 1/3 (crash) faulty servers, and can protect itself against denial of service attacks.

The contribution of this paper is the insight that we can tolerate Byzantine clients, when we make the assumption that servers are benign faulty. We believe this is a realistic assumption, since client software is often more exposed, while servers are typically behind corporate firewalls and intrusion detection systems, and thus better protected from compromise.

⋆ This work was performed while Alessandro Mei was a Marie Curie Fellow at the Computer Science and Engineering Department, University of California San Diego, USA. The fellowship is funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 253461.

2 Contribution: BP Fast Paxos

In BP Fast Paxos consensus is reached by the interaction between proposers and acceptors, and learners that learn the consensus value. Given our failure model, the mapping of these roles to clients and servers have two obvious choices: (1) clients are proposers, and servers are acceptors and learners, and (2) proposers are separate from both clients and servers. In the latter case, proposers interact with clients, and can be placed at the edge of a data center, and is thus more exposed to compromise than servers inside the data center.

We develop BP Fast Paxos by modifying Paxos in three essential ways: (i) The ⟨Prepare⟩ message sent by a proposer is replaced with a ⟨TrustChange⟩ message sent by acceptors. The reason for this change is that the acceptors can then detect misbehavior of a Byzantine proposer, and simply change its trust to another proposer whom can conclude the protocol. (ii) We also require that ⟨Accept⟩ messages (except those sent in round 0) contain a *proof* that it is legitimate. To construct this proof, a proposer must collect signed ⟨Promise⟩ messages from acceptors, so that acceptors can verify the proof against the value of the ⟨Accept⟩ message. (iii) Finally, we introduce a mechanism to *detect equivocation* by the proposer for the current round. That is, if the current proposer sends different ⟨Accept⟩ messages to different acceptors, then we can detect this using mechanisms similar to those used in Fast Paxos [3] to detect a collision in a fast round. This change requires that acceptors are also learners, enabling them to detect misbehavior through ⟨Learn⟩ messages, and replace a misbehaving proposer through a *trust change.*

In the normal case behavior of the algorithm, where no agents are faulty, a single unique proposer will try to have its value accepted by the acceptors. This initial accept does not contain any signatures, and consensus will complete in two communication steps, if no failures occur. Misbehavior is detected by learners (and thus acceptors), in which case a trust change is required. In this case, also HMAC-based signatures are necessary. If misbehavior or a crash is detected, then more rounds are necessary.

Many protocols have been derived from Paxos, however, BP Fast Paxos is the first protocol to leverage the separation of agent roles to distinguish their individual failure assumptions. The protocol is two-step and is safe even when all proposers are Byzantine, and does not require signatures for the common case.

References

1. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)
2. Lamport, L.: The part-time parliment. ACM Trans. on Comp. Syst. 16(2), 133–169 (1998)
3. Lamport, L.: Fast paxos. Distributed Computing 19(2), 79–103 (2006)
4. Mao, Y., Junqueira, F., Marzullo, K.: Towards low latency state machine replication for uncivil wide-area networks. In: Fifth Workshop on Hot Topics in Dependable Systems (HotDep 2009), Estoril, Lisbon, Portugal (June 2009)

Brief Announcement: On the Meaning of Solving a Task with a Failure Detector

Carole Delporte-Gallet[1], Hugues Fauconnier[1], Eli Gafni[2], and Petr Kuznetsov[3]

[1] University Paris Diderot
[2] Computer Science Department, University of California, Los Angeles, USA
[3] TU Berlin/Deutsche Telekom Laboratories, Berlin, Germany

We amend the framework, of two decades, of failure detectors [3,4] to bring it in line with the modern view of solving a distributed task [8] that separates *processes* and *threads*. While the conventional framework precludes a thread from advancing in the absence of failure detector values to "its" process, we allow live processes to advance the threads of failed processes. This provides for the application of the wealth of simulation techniques [2,6,7] designed for read-write threads and consequently to completely characterize task solvability with failure detectors. When dealing with the extremes, consensus and set-consensus, the former framework sufficed. With the advances in understanding of more nuanced notions like k-set consensus the framework requires amendment.

What does it mean to solve a task? Paxos state-machine replication protocol [8] proposes the notion of solvability in which every process is split into a *proposer* that submits commands to be executed, an *acceptor* that takes care of the command execution order, and a *learner* that receives the outcomes of executed commands. A command thus can still be executed when its proposer is slow or crashed.

The separation of computation from control leverages *simulation-based* computing proved to be very efficient in multiple contexts (e.g., [2,6,7]). Processes may help each other to make progress by simulating each others' steps. Even if a participating process has crashed, its thread may nevertheless be executed.

Alas, when it comes to solving a task using a *failure detector* [3,4], we cannot employ simulations. In a FD-based algorithm, each process periodically queries its *personal* FD module to get hints about failures of other processes. It then allowed to advance its personal thread rather than any thread for which input is available. It thus precludes "helping." But by enriching our model with a FD, we should expand our horizons rather than narrow them!

This paper changes the FD theory in a technically minute but consequential way. It tempers just with the definition of what does it *mean* to solve a task with a FD, to allow the processes to simulate each other threads. In our definition, the set of processes is partitioned into two classes: *synchronization* processes and *computation* processes. Computation processes solve tasks by receiving inputs and producing outputs. Synchronization processes help coordinating computation processes by using a FD. The FD provides each synchronization process with hints about the failures of other synchronization processes. The two classes of processes communicate by reading and writing in the shared memory. Since our framework allows for a simulation of computation processes, the power of FDs to solve tasks can be completely characterized.

Consider any task T. As any T can be solved 1-concurrently, i.e., assuming that at each moment of time there is at most one undecided participating process, there exists the maximal k such that T is solvable k-concurrently. We show that a FD $\mathcal{D}$ can be used to solve T if and only if $\mathcal{D}$ can be used to solve k-set agreement.

The conclusion is that a task is completely characterized through the "level of concurrency" its solution can tolerate. All tasks that can be solved k-concurrently but not $(k+1)$-concurrently (e.g., k-set agreement) are equivalent in the sense that they require exactly the same amount of information about failures. This characterization covers *all* tasks, including "colored" ones evading any characterization so far [6,1].

Delporte et al. [5] showed that any FD that allows for solving consensus among every pair of processes, also allows for solving consensus among *all* n processes. Years of trying to extend the result to k-set agreement ($k > 1$) bore no fruits. Using our new definition of solving a task, we immediately derive an even stronger result: a FD solving k-set agreement among an arbitrary given set of $k+1$ processes, can solve k-set agreement among all n processes.

The natural derivation of these results suggests that our FD framework captures the right way of thinking about FD-based distributed computations.

References

1. Afek, Y., Nir, I.: Failure detectors in loosely named systems. In: PODC, pp. 65–74. ACM Press, New York (2008)
2. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: STOC, pp. 91–100. ACM Press, New York (1993)
3. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
5. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Tight failure detection bounds on atomic object implementations. J. ACM 57(4) (2010)
6. 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)
7. Gafni, E., Kuznetsov, P.: Turning adversaries into friends: Simplified, made constructive, and extended. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 380–394. Springer, Heidelberg (2010)
8. Lamport, L.: The Part-Time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)

Brief Announcement: Algorithmic Mechanisms for Internet-Based Computing under Unreliable Communication*

Evgenia Christoforou[1], Antonio Fernández Anta[2,3], Chryssis Georgiou[1], and Miguel A. Mosteiro[4,3]

[1] University of Cyprus, Cyprus
{cs05ec1,chryssis}@cs.ucy.ac.cy
[2] Institute IMDEA Networks, Spain
antonio.fernandez@imdea.org
[3] Universidad Rey Juan Carlos, Spain
[4] Rutgers University, USA
mosteiro@cs.rutgers.edu

Abstract. This work, using a game-theoretic approach, considers Internet-based computations, where a master processor assigns, over the Internet, a computational task to a set of untrusted worker processors, and collects their responses. In particular, we consider a framework where altruistic, malicious, and rational workers co-exist, the communication between the master and the workers is not reliable, and that workers could be unavailable. Within this framework, we design algorithmic mechanisms that provide appropriate incentives to rational workers to act correctly, despite the malicious' workers actions and the unreliability of the network.

1 Motivation and Prior Work

In [1], an Internet-based master-worker framework was considered where a master processor assigns, over the Internet, a computational task to a set of untrusted worker processors and collects their responses. Three type of workers were assumed: *altruistic, malicious,* and *rational.* Altruistic workers always compute and return the correct result of the task, malicious workers always return an incorrect result, and rational (selfish) workers act based on their self interest. In other words, the altruistic and malicious workers have a predefined behavior: the first are *honest* and the latter are *cheaters* (they do not care about their utilities). Rational workers decide to be honest or to cheat based on which strategy would increase their utility. Under this framework, a game-theoretic mechanism

* This work is supported in part by the Cyprus Research Promotion Foundation grant ΤΠΕ/ΠΛΗΡΟ/0609(ΒΕ)/05, Comunidad de Madrid grant S2009TIC-1692, Spanish MICINN grant TIN2008–06735-C02-01, and the National Science Foundation grant CCF-0937829.

was designed that provided necessary incentives to the rational workers to compute and report the correct task result despite the malicious workers' actions. The design objective of the mechanism is for the master to force a desired *Nash Equilibrium* (NE), i.e., a strategy choice by each rational worker such that none of them has incentive to change it. That NE is the one in which the master achieves a desired probability of obtaining the correct task result, while maximizing its benefit. The utility of the mechanism was demonstrated by applying it to two paradigmatic applications: a SETI-like volunteer computing system and a contractor-based system, such as Amazon's mechanical turk. This work has not considered the possibility of network unreliability, which is a factor that cannot be ignored in Internet-based computations [2].

2 Contributions

This work extends the master-worker framework of [1] by additionally considering the possibility that the communication between the master and the workers is not reliable. That is, we consider the possibility that messages exchanged may get lost or arrive late. This communication uncertainty can either be due to communication-related failures or due to workers being slow in processing messages (or even crashing while doing so). For instance, Heien at al. [2] have found that in BOINC only around 5% of the workers are available more than 80% of the time, and that half of the workers are available less than 40% of the time. This fact, combined with the length of the computation incurred by a task [3], justifies the interest of considering in the Internet-based master-worker framework the possibility of workers not replying. In order to introduce this possibility in the framework, we consider that there is some positive probability that the master does not receive a reply from a given worker. Since it is now possible for a worker's reply not to reach the master, we additionally extend the framework of [1] by allowing workers to abstain from the computation. Imagine the situation where a rational worker decides to compute and truthfully return the task result but its reply is not received by the master. In this case the master provides no reward to the worker, while the worker has incurred the cost of performing the task. Hence, it is only natural to provide to the workers the choice of not replying, especially when the reliability of the network is low. This makes the task of the master even more challenging, as it needs to provide the necessary incentives to encourage rational workers to reply and do so truthfully, even in the presence of low network reliability.

Within this extended framework, we develop and analyze two game-theoretic mechanisms, a time-based mechanism and a reply-based one, that provide the necessary incentives for the rational workers to truthfully compute and return the task result, despite the malicious workers' actions and the network unreliability. Furthermore, we apply our mechanisms to two realistic settings: SETI-like volunteer computing applications and contractor-based applications such as Amazon's mechanical turk. *Full details can be found in [4].*

References

1. Fernández Anta, A., Georgiou, C., Mosteiro, M.A.: Algorithmic Mechanisms for Internet-based Master-Worker Computing with Untrusted and Selfish Workers. In: Proc. of IPDPS 2010, pp. 378–388 (2010)
2. Heien, E.M., Anderson, D.P., Hagihara, K.: Computing low latency batches with unreliable workers in volunteer computing environments. Journal of Grid Computing 7, 501–518 (2009)
3. Kondo, D., Araujo, F., Malecot, P., Domingues, P., Silva, L., Fedak, G., Cappello, F.: Characterizing result errors in internet desktop grids. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 361–371. Springer, Heidelberg (2007)
4. Technical Report of this work, http://www.cs.ucy.ac.cy/ric/MARfailTR.pdf

Maximum Metric Spanning Tree Made Byzantine Tolerant*

Swan Dubois[1], Toshimitsu Masuzawa[2], and Sébastien Tixeuil[3]

[1] UPMC Sorbonne Universités & INRIA, France
[2] Osaka University, Japan
[3] UPMC Sorbonne Universités & Institut Universitaire de France, France

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. This paper focuses on systems that are both self-stabilizing and Byzantine tolerant.

We consider the well known problem of constructing a maximum metric tree in this context. Combining these two properties is known to induce many impossibility results. In this paper, we first provide two new impossibility results about the construction of a maximum metric tree in presence of transient and (permanent) Byzantine faults. Then, we propose a new self-stabilizing protocol that provides optimal containment to an arbitrary number of Byzantine faults.

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] 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 [5,6,7] as impossibility results are expected in many cases.

Related Works. A promising path towards multi-tolerance to both transient and Byzantine faults is *Byzantine containment*. For *local* tasks (*i.e.* tasks whose correctness can be checked locally, such as vertex coloring, link coloring, or dining

* This work has been supported in part by ANR projects SHAMAN and SPADES, by MEXT Global COE Program and by JSPS Grant-in-Aid for Scientific Research ((B) 22300009).

philosophers), the notion of *strict stabilization* was proposed [7,8]. 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. To circumvent impossibility results, the weaker notion of *strong stabilization* was proposed [9,10]: here, correct nodes outside the containment radius may be perturbed by the actions of Byzantine nodes, but only a finite number of times.

Recently, the idea of generalizing the containment area in strict and strong stabilization to an area that depends both on the graph topology and the problem to be solved rather than an arbitrary fixed containment radius was proposed [11,12] and denoted by *topology aware* strict (and strong) stabilization. When maximizable metric trees are considered, [11] proposed an optimal (with respect to impossibility results) protocol for topology-aware strict stabilization, and for the simpler case of breath-first-search trees, [12] presented a protocol that is optimal both with respect to strict and strong variants of topology-aware stabilization. Up to this paper, the case of optimality for topology-aware strong stabilization in the general maximal metric case remained open.

Our Contribution. In this paper, we investigate the possibility of topology-aware strong stabilization for tasks that are global (*i.e.* there exists a causality chain of size r, where r depends on n the size of the network), and focus on the maximum metric tree problem. Our contribution in this paper is threefold. First, we provide two impossibility results for self-stabilizing maximum metric tree construction in presence of Byzantine faults. In more details, we characterize a specific class of maximizable metrics (that includes breath-first-search and shortest path metrics) that prevents the existence of strong stabilizing solutions and we provide a lower bound on the containment area for topology-aware strong stabilization (Section 3). Second, we provide a topology-aware strongly stabilizing protocol that matches this lower bound on the containment area (Section 4). Finally, we provide a necessary and sufficient condition for the existence of a strongly stabilizing solution (Section 5).

2 Model, Definitions and Previous Results

2.1 State Model

A *distributed system* $S = (V, L)$ consists of a set $V = \{v_1, v_2, \ldots, v_n\}$ of processes and a set L 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 L, so we use graph terminology to describe a distributed system S. We use the following notations: $n = |V|$, $m = |L|$ and $d(u, v)$ denotes the distance between two processes u and v (*i.e* the length of the shortest path between u and v). Processes u and

v are called *neighbors* if $(u, v) \in L$. The set of neighbors of a process v is denoted by N_v. We assume each process can distinguish its neighbors from each other by locally labeling them. We denote the maximal degree of the system by Δ.

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 identical. 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 action executed by each process is 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. 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'.

We say that a process is *enabled* in a configuration ρ if the guard of at least one of its actions is evaluated as true in 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 distinguish some properties on the scheduler (or daemon). A *distributed daemon* schedules the actions of processes such that any subset of processes can simultaneously execute their actions. We say that the daemon is *central* if it schedules action of only one process at any step. 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 but we add the following assumption on schedules: any schedule is strongly fair (that is, it is impossible for any process to be infinitely often enabled without executing its action infinitely often in an execution) and k-bounded (that is, it is impossible for any process to execute more than k actions between two consecutive actions of any other process).

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. If v is a Byzantine process, v can repeatedly change its variables arbitrarily. For a given execution, the number of faulty processes is arbitrary but we assume that the root process is never faulty.

2.2 Self-Stabilizing Protocols 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 *specification predicate* (shortly, specification), $spec(v)$, for each process v: a configuration is a desired one (with a solution) if every process satisfies $spec(v)$. A specification $spec(v)$ is a boolean expression on variables of P_v ($\subseteq V$) where P_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 specification $spec(v)$.

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, *i.e.* when 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 [7] define a *strictly stabilizing protocol* as a self-stabilizing protocol resilient to unbounded number of Byzantine processes. More formally, given an integer c, a process is c-correct if it is correct (*i.e.* not Byzantine) and located at distance more than c from any Byzantine process. 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 refers to the *containment radius* defined in [7]. The parameter f refers explicitly to the number of Byzantine processes, while [7] dealt with unbounded number of Byzantine faults (that is $f \in \{0 \dots n\}$).

Definition 1 ((c, f)-strict stabilization). *A protocol is* (c, f)-strictly stabilizing *for specification spec if, given at most f Byzantine processes, any execution $e = \rho_0, \rho_1, \dots$ contains a configuration ρ_i that is (c, f)-contained for spec.*

Strong stabilization. To circumvent impossibility results related to strict stabilization, [10] defines a weaker notion. Here, the requirement to the containment radius is relaxed, *i.e.* there may exist processes outside the containment radius that invalidate the specification predicate, due to Byzantine actions. However, the impact of Byzantine triggered action is limited in times: the set of Byzantine processes may only impact processes outside the containment radius a bounded number of times, even if Byzantine processes execute an infinite number of actions.

More formally, [10] defines strong stabilization as follows. From the states of c-correct processes, *c-legitimate configurations* and *c-stable configurations* are defined as follows. A configuration ρ is c-legitimate for *spec* if every c-correct process v satisfies *spec*(v). A configuration ρ is c-stable if every c-correct process never changes the values of its O-variables as long as Byzantine processes make no action. Roughly speaking, the aim of self-stabilization is to guarantee that a distributed system eventually reaches a c-legitimate and c-stable configuration. However, a self-stabilizing system can be disturbed by Byzantine processes after reaching a c-legitimate and c-stable configuration. The *c-disruption* represents the period where c-correct processes are disturbed by Byzantine processes and is defined as follows. A portion of execution $e = \rho_0, \rho_1, \ldots, \rho_t$ $(t > 1)$ is a c-disruption if and only if the following holds: (i) e is finite, (ii) e contains at least one action of a c-correct process for changing the value of an O-variable, (iii) ρ_0 is c-legitimate for *spec* and c-stable, and (iv) ρ_t is the first configuration after ρ_0 such that ρ_t is c-legitimate for *spec* and c-stable. A configuration ρ_0 is (t, k, c, f)-time contained for *spec* if given at most f Byzantine processes, the following properties are satisfied: (i) ρ_0 is c-legitimate for *spec* and c-stable, (ii) every execution starting from ρ_0 contains a c-legitimate configuration for *spec* after which the values of all the O-variables of c-correct processes remain unchanged (even when Byzantine processes make actions repeatedly and forever), (iii) every execution starting from ρ_0 contains at most t c-disruptions, and (iv) every execution starting from ρ_0 contains at most k actions of changing the values of O-variables for each c-correct process.

Definition 2 ((t, c, f)-strongly stabilizing protocol). *A protocol A is (t, c, f)-strongly stabilizing if and only if starting from any arbitrary configuration, every execution involving at most f Byzantine processes contains a (t, k, c, f)-time contained configuration. Parameter k is the (t, c, f)-process-disruption times of A.*

Topology-aware Byzantine resilience. We describe here another weaker notion than the strict stabilization: the *topology-aware strict stabilization* (denoted by TA strict stabilization for short) introduced by [11]. Here, the requirement to the containment radius is relaxed, *i.e.* the set of processes that may be disturbed by Byzantine ones is not reduced to the union of c-neighborhood of Byzantine processes (*i.e.* the set of processes at distance at most c from a Byzantine process) but can be defined depending on the graph topology and Byzantine processes' locations.

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 (that is function of B) denotes a subset of V (intuitively, this set gathers all processes that may be disturbed by Byzantine processes). A process is *S_B-correct* if it is a correct process (*i.e.* not Byzantine) that does not belongs to S_B. A configuration ρ is *S_B-legitimate* for *spec* if every S_B-correct process v is legitimate for *spec* (*i.e.* if *spec*(v) holds). A configuration ρ_0 is *(S_B, f)-TA 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 refers to the *containment area*. Any process that belongs to this set may be infinitely disturbed by Byzantine processes. The parameter f refers explicitly to the number of Byzantine processes.

Definition 3 ((S_B, f)-TA strict stabilization). *A protocol is* (S_B, f)-TA 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)-TA contained for spec.*

Similarly to topology-aware strict stabilization, we can weaken the notion of strong stabilization using the notion of containment area. This idea was introduced by [12]. We recall in the following the formal definition of this concept. A configuration ρ is S_B-stable if every S_B-correct process never changes the values of its O-variables as long as Byzantine processes make no action. A portion of execution $e = \rho_0, \rho_1, \ldots, \rho_t$ $(t > 1)$ is a S_B-TA disruption if and only if the followings hold: (i) e is finite, (ii) e contains at least one action of a S_B-correct process for changing the value of an O-variable, (iii) ρ_0 is S_B-legitimate for *spec* and S_B-stable, and (iv) ρ_t is the first configuration after ρ_0 such that ρ_t is S_B-legitimate for *spec* and S_B-stable. A configuration ρ_0 is (t, k, S_B, f)-TA time contained for *spec* if given at most f Byzantine processes, the following properties are satisfied: (i) ρ_0 is S_B-legitimate for *spec* and S_B-stable, (ii) every execution starting from ρ_0 contains a S_B-legitimate configuration for *spec* after which the values of all the O-variables of S_B-correct processes remain unchanged (even when Byzantine processes make actions repeatedly and forever), (iii) every execution starting from ρ_0 contains at most t S_B-TA disruptions, and (iv) every execution starting from ρ_0 contains at most k actions of changing the values of O-variables for each S_B-correct process.

Definition 4 ((t, S_B, f)-TA strongly stabilizing protocol). *A protocol A is (t, S_B, f)-TA strongly stabilizing if and only if starting from any arbitrary configuration, every execution involving at most f Byzantine processes contains a (t, k, S_B, f)-TA time contained configuration that is reached after at most l rounds. Parameters l and k are respectively the (t, S_B, f)-stabilization time and the (t, S_B, f)-process disruption times of A.*

2.3 Maximum Metric Tree Construction

In this work, we deal with maximum (routing) metric trees. 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 $\prec$. We recall all definitions and notations introduced in [13].

A *routing metric* (or just *metric*) is a five-tuple $(M, W, met, mr, \prec)$ where: (i) M is a set of metric values, (ii) W is a set of edge weights, (iii) met is a metric function whose domain is $M \times W$ and whose range is M, (iv) mr is the maximum metric value in M with respect to $\prec$ and is assigned to the root of the system, and (v) $\prec$ is a less-than total order relation over M. The $\prec$ relation must satisfy the following three conditions for arbitrary metric values

m, m', and m'' in M: (i) irreflexivity ($m \not\prec m$), (ii) transitivity (if $m \prec m'$ and $m' \prec m''$ then $m \prec m''$), and (iii) totality ($m \prec m'$ or $m' \prec m$ or $m = m'$). Any metric value $m \in M \setminus \{mr\}$ must satisfy 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, this model allows to modelize the following metrics: the shortest path metric ($\mathcal{SP}$), the flow metric ($\mathcal{F}$), and the reliability metric ($\mathcal{R}$) as pointed in [13]. Note also that we can modelize the construction of a spanning tree with no particular constraints or a BFS spanning tree using a routing metric (respectively denoted by $\mathcal{NC}$ and by $\mathcal{BFS}$).

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. 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 $\forall i \in \{1, \ldots, k-1\}, m_i = met(m_{i-1}, wf(\{v_i, v_{i-1}\}))$ and $m_0 = mr$. 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 $\prec$. 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. 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 [13] is the study of metrics that always allow the construction of a maximum metric tree. More formally, the definition follows. 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.

Given a maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$, the aim of this work is to study the construction of a maximum metric tree with respect to $\mathcal{M}$ that spans the system in a self-stabilizing way in a system subject to permanent Byzantine faults (but we assume that the root process is never a Byzantine one). 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 ($prnt_v \in N_v \cup \{\bot\}$), a level that stores its current metric value ($level_v \in M$) and an integer that stores a distance ($dist_v \in \mathbb{N}$). 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 neighbors, we denote by $w_{u,v}$ the weight of the edge $\{u,v\}$ (that is, the value of $wf(\{u,v\})$).

Definition 5 ($\mathcal{M}$-path). *Given an assigned metric $\mathcal{M} = (M, W, met, mr, \prec, wf)$ over a system S, a path $(v_0, \dots, v_k)$ $(k \geq 1)$ of S is a $\mathcal{M}$-*path *if and only if: (i) $prnt_{v_0} = \bot$, $level_{v_0} = mr$, $dist_{v_0} = 0$, and $v_0 \in B \cup \{r\}$, (ii) $\forall i \in \{1, \dots, k\}, prnt_{v_i} = v_{i-1}$ and $level_{v_i} = met(level_{v_{i-1}}, w_{v_i,v_{i-1}})$, (iii) $\forall i \in \{1, \dots, k\}, met(level_{v_{i-1}}, w_{v_i,v_{i-1}}) = max_{\prec}\{met(level_u, w_{v_i,u}) | u \in N_{v_i}\}$, (iv) $\forall i \in \{1, \dots, k\}, dist_{v_i} = legal\_dist_{v_{i-1}}$ (with $\forall u \in N_v, legal\_dist_u = dist_u + 1$ if $level_v = level_u$ and $legal\_dist_u = 0$ otherwise), and (v) $level_{v_k} = \mu(v_k, v_0)$.*

We define the specification predicate $spec(v)$ of the maximum metric tree construction with respect to a maximizable metric $\mathcal{M}$ as follows.

$$spec(v) : \begin{cases} prnt_v = \bot \text{ and } level_v = mr, \text{ and } dist_v = 0 \text{ if } v \text{ is the root } r \\ \text{there exists a } \mathcal{M}\text{-path } (v_0, \dots, v_k) \text{ such that } v_k = v \text{ otherwise} \end{cases}$$

2.4 Previous Results

The first interesting result is due to [13] that provides a full characterization of maximizable metrics as follow. 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 $met(m,w) = m$. 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 $met(m,w) = met(m',w))$. Then, [13] proves that a metric is maximizable if and only if this metric is bounded and monotonic. Secondly, [14] provides a self-stabilizing protocol to construct a maximum metric tree with respect to any maximizable metric.

Now, we focus on self-stabilizing solutions resilient to Byzantine faults. Following [7], it is obvious that there exists no strictly stabilizing protocol for this problem. If we consider the weaker notion of topology-aware strict stabilization, [11] defines the best containment area as: $S_B = \{v \in V \setminus B | \mu(v,r) \preceq max_{\prec} \{\mu(v,b) | b \in B\}\} \setminus \{r\}$. Intuitively, S_B gathers correct processes that are closer (or at equal distance) from a Byzantine process than the root according to the metric. Moreover, [11] proves that the algorithm introduced for the maximum metric spanning tree construction in [14] achieves this optimal containment area. In other words, [11] proves the following results: (i) given a maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$, 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 $A_B \subsetneq S_B$ and (ii) given a maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$, the protocol of [14] is a $(S_B, n-1)$-TA strictly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{M}$.

Some other works try to circumvent the impossibility result of strict stabilization using the concept of strong stabilization but do not provide results for all maximizable metric. Indeed, [10] proves the following result about spanning trees: there exists a $(t, 0, n-1)$-strongly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{NC}$ (that is, for a spanning tree with no particular constraints) with a finite t. On the other hand, regarding BFS spanning tree construction, [12] proved the following impossibility result: even under the central daemon, there exists no $(t, c, 1)$-strongly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{BFS}$ where t and c are two finite integers.

Now, if we focus on topology-aware strong stabilization, [12] introduced the following containment area: $S_B^* = \{v \in V | min\{d(v,b)|b \in B\} < d(r,v)\}$. We proved then the following results: (i) even under the central daemon, there exists no $(t, A_B^*, 1)$-TA strongly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{BFS}$ where $A_B^* \subsetneq S_B^*$ and t is a finite integer and (ii) the protocol of [15] is a $(t, S_B^*, n-1)$-TA strongly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{BFS}$ where t is a finite integer.

The main motivation of this work is to fill the gap between results about TA strong and strong stabilization in the general case (that is, for any maximizable metric). Mainly, we define the best possible containment area for TA strong stabilization, we propose a protocol that provides this containment area and we characterize the set of metrics that allow strong stabilization.

3 Impossibility Results

We provide here our impossibility results about containment radius (resp. area) of any strongly stabilizing (resp. TA strongly stabilizing) protocol for the maximum metric tree construction.

Strong Stabilization. We introduce here new definitions to characterize some important properties of maximizable metrics that are used in the following. A metric $\mathcal{M} = (M, W, met, mr, \prec)$ is *strictly decreasing* if, for any metric value $m \in M$, the following property holds: either $\forall w \in W, met(m,w) \prec m$ or $\forall w \in W, met(m,w) = m$. A metric value m is a *fixed point* of a metric $\mathcal{M} = (M, W, met, mr, \prec)$ if $m \in M$ and if for any value $w \in W$, we have: $met(m,w) = m$. Then, we define a specific class of maximizable metrics and we prove that it is impossible to construct a maximum metric tree in a strongly-stabilizing way if the considered metric is not in the class.

Definition 6 (Strongly maximizable metric). *A maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$ is strongly maximizable if and only if $|M| = 1$ or if the following properties hold: (i) $|M| \geq 2$, (ii) $\mathcal{M}$ is strictly decreasing, and (iii) $\mathcal{M}$ has one and only one fixed point.*

Note that $\mathcal{NC}$ is a strongly maximizable metric (since $|M| = 1$) whereas $\mathcal{BFS}$ or $\mathcal{F}$ are not (since the first one has no fixed point, the second is not strictly decreasing). Now, we can state our first impossibility result (the proof is available in [16]).

Theorem 1. *Given a maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$, even under the central daemon, there exists no $(t, c, 1)$-strongly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{M}$ for any finite integer t if: (i) $\mathcal{M}$ is not a strongly maximizable metric, or (ii) $c < |M| - 2$.*

Topology Aware Strong Stabilization. First, we generalize the set S_B^* previously defined for the $\mathcal{BFS}$ metric in [12] to any maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$. From now, S_B^* denotes the following set:

$$S_B^* = \{v \in V \setminus B \,|\mu(v, r) \prec max_{\prec}\{\mu(v, b)|b \in B\}\}$$

Intuitively, S_B^* gathers the set of correct processes that are strictly closer (according to $\mathcal{M}$) to a Byzantine process than the root. Note that we assume for the sake of clarity that $V \setminus S_B^*$ induces a connected subsystem. If it is not the case, then S_B^* is extended to include all processes belonging to connected subsystems of $V \setminus S_B^*$ that do not contain r. Now, we can state our generalization of impossibility result of [12] (the proof is available in [16]).

Theorem 2. *Given a maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$, even under the central daemon, there exists no $(t, A_B^*, 1)$-TA-strongly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{M}$ where $A_B^* \subsetneq S_B^*$ and t is a given finite integer.*

4 Topology-Aware Strongly Stabilizing Protocol

The goal of this section is to provide a $(t, S_B^*, n-1)$-TA strongly stabilizing protocol in order to match the lower bound on containment area provided by Theorem 2. If we focus on the protocol provided by [11] (that is $(S_B, n-1)$-TA strictly stabilizing), we can prove that this protocol does not satisfy our constraints since it is not $(t, S_B^*, 2)$-TA strongly stabilizing (see [16]).

4.1 Presentation of the Protocol

Our protocol needs a supplementary assumption. We introduce the following definition.

Definition 7 (Set of used metric values). *Given an assigned metric $\mathcal{AM} = (M, W, met, mr, \prec, wf)$ over a system S, the* set of used metric values *of $\mathcal{AM}$ is defined as $M(S) = \{m \in M | \exists v \in V, (\mu(v, r) = m) \vee (\exists b \in B, \mu(v, b) = m)\}$.*

We assume that we always have $|M(S)| \geq 2$ (the necessity of this assumption is explained below). Nevertheless, note that the contrary case ($|M(S)| = 1$) is possible if and only if the assigned metric is equivalent to $\mathcal{NC}$. As the protocol of [10] performs $(t, 0, n-1)$- strong stabilization with a finite t for this metric, we can achieve the $(t, S_B^*, n-1)$-TA strong stabilization when $|M(S)| = 1$ (since this implies that $S_B^* = \emptyset$). In this way, this assumption does not weaken the possibility result.

Although the protocol of [11] is not $(t, S_B^*, n-1)$-TA strongly stabilizing, our protocol borrows fundamental strategy from it. In this protocol, any process tries to maximize its *level* in the tree by choosing as its parent the neighbor that provides the best metric value. The key idea of this protocol is to use the *dist* variable (upper bounded by a given constant D) to detect and break cycles of processes that have the same maximum metric. To achieve $(S_B, n-1)$-TA strict stabilization, the protocol ensures a fair selection along the set of its neighbors with a round-robin order.

The possibility of infinite number of S_B^*-TA disruptions of the protocol of [11] mainly comes from the following fact: a Byzantine process can independently lie about its *level* and its *dist* variables. For example, a Byzantine process can provide a *level* equal to mr and a *dist* arbitrarily large. In this way, it may lead a correct process of $S_B \setminus S_B^*$ to have a *dist* variable equal to $D-1$ such that no other correct process can choose it as its parent (this rule is necessary to break cycle) but it cannot modify its state (this rule is only enabled when *dist* is equal to D). Then, this process may always prevent some of its neighbors to join a $\mathcal{M}$-path connected to the root and hence allow another Byzantine process to perform an infinite number of S_B^*-TA disruptions.

It is why we modified the management of the *dist* variable (note that other variables are managed exactly in the same way as in the protocol of [11]). In order to contain the effect of Byzantine processes on *dist* variables, each process that has a *level* different from the one of its parent in the tree sets its *dist* variable to 0. In this way, a Byzantine process modifying its *dist* variable can only affect correct processes that have the same *level*. Consequently, in the case where $|M(S)| \geq 2$, we are ensured that correct processes of $S_B \setminus S_B^*$ cannot keep a *dist* variable equal or greater than $D-1$ infinitely. Hence, a correct process of $S_B \setminus S_B^*$ cannot be disturbed infinitely often without joining a $\mathcal{M}$-path connected to the root.

We can see that the assumption $|M(S)| \geq 2$ is essential to perform the TA strong stabilization. Indeed, in the case where $|M(S)| = 1$, Byzantine processes can play exactly the scenario described above (in this case, our protocol is equivalent to the one of [11]).

The second modification we bring to the protocol of [11] follows. When a process has an inconsistent *dist* variable with its parent, we allow it only to increase its *dist* variable. If the process needs to decrease its *dist* variable (when it has a strictly greater distance than its parent), then it must change its parent. This rule allows us to bound the maximal number of steps of any process between two modifications of its parent (a Byzantine process cannot lead a correct one to infinitely often increase and decrease its distance without modifying its parent).

Our protocol is formally described in Algorithm 4.1.

4.2 Proof of the $(t, S_B^*, n-1)$-TA Strong Stabilization for *spec*

Due to the lack of space, only key ideas of this proof are provided but complete proofs are available in [16]. First, we prove the following result with a proof very similar to the one of [11].

Algorithm 4.1. *$\mathcal{SSMAX}$*, TA strongly 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:
    $prnt_v \in \begin{cases} \{\bot\} \text{ if } v = r \\ N_v \text{ if } v \neq r \end{cases}$ : pointer on the parent of $v$ in the tree.
    $level_v \in M$: metric of the node.
    $dist_v \in \{0, \dots, D\}$: hop counter.
Functions:
    For any subset $A \subseteq N_v$, $choose_v(A)$ returns the first element of $A$ that is bigger than $prnt_v$
    (in a round-robin fashion).
    $current\_dist_v() = \begin{cases} 0 \text{ if } level_{prnt_v} \neq level_v \\ min\{dist_{prnt_v} + 1, D\} \text{ if } level_{prnt_v} = level_v \end{cases}$
Rules:
    $(\boldsymbol{R_r})$ :: $(v = r) \wedge ((level_v \neq mr) \vee (dist_v \neq 0)) \longrightarrow level_v := mr;\ dist_v := 0$
    $(\boldsymbol{R_1})$ :: $(v \neq r) \wedge (prnt_v \in N_v) \wedge ((dist_v < current\_dist_v()) \vee$
            $(level_v \neq met(level_{prnt_v}, w_{v,prnt_v})))$
            $\longrightarrow level_v := met(level_{prnt_v}, w_{v,prnt_v});\ dist_v := current\_dist_v()$
    $(\boldsymbol{R_2})$ :: $(v \neq r) \wedge ((dist_v = D) \vee (dist_v > current\_dist_v())) \wedge (\exists u \in N_v, dist_u < D - 1)$
            $\longrightarrow prnt_v := choose_v(\{u \in N_v | dist_u < D-1\});\ level_v := met(level_{prnt_v}, w_{v,prnt_v});$
            $dist_v := current\_dist_v()$
    $(\boldsymbol{R_3})$ :: $(v \neq r) \wedge (\exists u \in N_v, (dist_u < D - 1) \wedge (level_v \prec met(level_u, w_{u,v})))$
            $\longrightarrow prnt_v := choose_v(\{u \in N_v | (level_u < D - 1) \wedge$
            $(met(level_u, w_{u,v}) = max_{\prec}\{met(level_q, w_{q,v}) | q \in N_v, level_q < D - 1\})\});$
            $level_v := met(level_{prnt_v}, w_{prnt_v,v});\ dist_v := current\_dist_v()$
```

Theorem 3. *$\mathcal{SSMAX}$ is a $(S_B, n-1)$-TA-strictly stabilizing protocol for spec.*

We introduce here some notations. Given a configuration $\rho \in C$ and a metric value $m \in M$, we define the following predicate: $IM_m(\rho) \equiv \forall v \in V, level_v \preceq max_{\prec}\{m, max_{\prec}\{\mu(v,u) | u \in B \cup \{r\}\}\}$. Given an assigned metric to a system S, we can observe that the set of metric values $M(S)$ is finite and that we can label elements of $M(S)$ by $m_0 = mr, m_1, \dots, m_k$ in a way such that $\forall i \in \{0, \dots, k-1\}, m_{i+1} \prec m_i$. Let $\mathcal{LC}$ be the following set of configurations:

$$\mathcal{LC} = \left\{\rho \in \mathcal{C} \middle| (\forall v \in V \setminus S_B, spec(v)) \wedge (IM_{m_k}(\rho))\right\}$$

Let $E_B = S_B \setminus S_B^*$ (*i.e.* E_B is the set of process v such that $\mu(v,r) = max\{\mu(v,b) | b \in B\}$). Note that the subsystem induced by E_B may have several connected components. In the following, we use the following notations: $E_B = \{E_B^1, \dots, E_B^\ell\}$ where each E_B^i ($i \in \{0, \dots, \ell\}$) is a subset of E_B inducing a maximal connected component, $\delta(E_B^i)$ ($i \in \{0, \dots, \ell\}$) is the diameter of the subsystem induced by E_B^i, and $\delta = max\{\delta(E_B^i) | i \in \{0, \dots, \ell\}\}$. When a and b are two integers, we define the following function: $\Pi(a,b) = \frac{a^{b+1}-1}{a-1}$.

Let ρ be a configuration of $\mathcal{LC}$ and e be an execution starting from ρ. Let p be a process of E_B^i ($i \in \{0, \dots, \ell\}$) such that there exists a neighbor q that satisfies $q \in V \setminus S_B$ and $\mu(p,r) = met(\mu(q,r), w_{p,q})$ (such a process exists by construction of E_B^i). Then, we can prove by induction the following property: if v is a process of E_B^i such that $d_{E_B^i}(p,v) = d$ (where $d_{E_B^i}$ denotes the distance in the subsystem induced by E_B^i), then v executes at most $\Pi(k,d)\Delta D$ actions in e. We can deduce the following lemma:

Lemma 1. *If ρ is a configuration of $\mathcal{LC}$, then any process $v \in E_B$ is activated at most $\Pi(k,\delta)\Delta D$ times in any execution starting from ρ.*

If we assume that there exists an execution starting from a configuration of $\mathcal{LC}$ such that a process $v \in E_B$ is never enabled and $spec(v)$ is infinitely often false, we can find a contradiction with the construction of the algorithm. Hence the following lemma holds:

Lemma 2. *If ρ is a configuration of $\mathcal{LC}$ and v is a process such that $v \in E_B$, then for any execution e starting from ρ either (i) there exists a configuration ρ' of e such that $spec(v)$ is always satisfied after ρ', or (ii) v is activated in e.*

Let $\mathcal{LC}^*$ be the following set of configurations:

$$\mathcal{LC}^* = \{\rho \in C | (\rho \text{ is } S_B^*\text{-legitimate for } spec) \wedge (IM_{m_k}(\rho) = true)\}$$

Note that, as $S_B^* \subseteq S_B$, we can deduce that $\mathcal{LC}^* \subseteq \mathcal{LC}$. Hence, properties of Lemmas 1 and 2 also apply to configurations of $\mathcal{LC}^*$ and we can deduce the following lemma:

Lemma 3. *Configurations of $\mathcal{LC}^*$ are $(n\Pi(k,\delta)\Delta D, \Pi(k,\delta)\Delta D, S_B^*, n-1)$-TA time contained for spec and any execution of $\mathcal{SSMAX}$ (starting from any configuration) contains such a configuration.*

Finally, we can deduce the main theorem:

Theorem 4. *$\mathcal{SSMAX}$ is a $(n\Pi(k,\delta)\Delta D, S_B^*, n-1)$-TA strongly stabilizing protocol for spec.*

5 Concluding Remarks

We now discuss the relationship between TA strong and strong stabilization for maximum metric tree construction. As a matter of fact, the set of assigned metrics that allow strong stabilization can be characterized. Indeed, properties about the metric itself are not sufficient to decide the possibility of strong stabilization: it is necessary to include some knowledge about the considered system (typically, the metric assignement). Informally, it is possible to construct a maximum metric tree in a strongly stabilizing way if and only if the considered metric is strongly maximizable and if the desired containment radius is sufficiently large with respect to the size of the set of used metric values. The formal statement of this result follows.

Theorem 5. *Given an assigned metric $\mathcal{AM} = (M, W, mr, met, \prec, wf)$ over a system S, there exists a $(t, c, n-1)$-strongly stabilizing protocol for maximum metric spanning tree construction with a finite t if and only if (i) $(M, W, met, mr, \prec)$ is a strongly maximizable metric, and (ii) $c \geq max\{0, |M(S)| - 2\}$.*

Table 1. Summary of results

	$\mathcal{M} = (M, W, mr, met, \prec)$ is a maximizable metric
(c, f)-strict stabilization	Impossible (for any c and f) by [7]
(t, c, f)-strong stabilization	Possible $\iff \begin{cases} \mathcal{M} \text{ is a strongly maximizable metric, and} \\ c \geq max\{0, \lvert M(S)\rvert - 2\} \end{cases}$ (for $0 \leq f \leq n-1$ and a finite t) by Theorem 5
(A_B, f)-TA strict stabilization	Impossible (for any f and $A_B \subsetneq S_B$) by [11]
(S_B, f)-TA strict stabilization	Possible (for $0 \leq f \leq n-1$) by [11] and Theorem 3
(t, A_B, f)-TA strong stabilization	Impossible (for any f and $A_B \subsetneq S_B^*$) by Theorem 2
(t, S_B^*, f)-TA strong stabilization	Possible (for $0 \leq f \leq n-1$ and a finite t) by Theorem 4

The "only if" part of this Theorem is a direct consequence of Theorem 1 when we observe that $|M(S)| \leq |M|$ by definition. The "if" part of this Theorem comes from the following observations. If $|M(S)| = 1$, it is equivalent to construct a maximum metric spanning tree for $\mathcal{M}$ and for $\mathcal{NC}$ over this system. By [10], we know that there exists a $(t, 0, n-1)$-strongly stabilizing protocol for this problem with a finite t, that proves the result. If $|M(S)| \geq 2$, we can prove that $S_B^* = \{v \in V | min\{d(v, b)|b \in B\} \leq c\}$ and then we have the result by Theorem 4.

We can now summarize all results about self-stabilizing maximum metric tree construction in presence of Byzantine faults with Table 1. Note that results provided in this paper close every open question raised in related works and that they subsume results from [10] and [12].

Our choice was to work with a specification of the problem that considers the *dist* variable as an O-variable. This choice may appear a bit strong but it permitted us to maintain consistency in the presentation of the results. In fact, impossibility results of Section 3 can be proved with a weaker specification that does not consider the *dist* variable as an O-variable (see [17]). On the other hand, the stronger specification eased the bounding of the number of disruptions of the proposed protocol. Our protocol is also TA strongly stabilizing with the weaker specification but we were not able to exactly bound the number of disruptions.

The following questions are still open. Is it possible to bound the number of disruptions for the weaker specification? Is it possible to perform TA strong stabilization using a weaker daemon requirement? Is it possible to decrease the number of disruptions without loosing containment optimality?

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
2. Dolev, S.: Self-stabilization. MIT Press, Cambridge (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. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. J. ACM 51(5), 780–799 (2004)

6. Daliot, A., Dolev, D.: Self-stabilization of byzantine protocols. In: Herman, T., Tixeuil, S. (eds.) SSS 2005. LNCS, vol. 3764, pp. 48–67. Springer, Heidelberg (2005)
7. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: 21st Symposium on Reliable Distributed Systems (SRDS 2002), p. 22. IEEE Computer Society, Los Alamitos (2002)
8. 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)
9. Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 440–453. Springer, Heidelberg (2006)
10. Dubois, S., Masuzawa, T., Tixeuil, S.: Bounding the impact of unbounded attacks in stabilization. IEEE Transactions on Parallel and Distributed Systems, TPDS (2011)
11. Dubois, S., Masuzawa, T., Tixeuil, S.: The impact of topology on byzantine containment in stabilization. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 495–509. Springer, Heidelberg (2010)
12. Dubois, S., Masuzawa, T., Tixeuil, S.: On byzantine containment properties of the min+1 protocol. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 96–110. Springer, Heidelberg (2010)
13. Gouda, M.G., Schneider, M.: Maximizable routing metrics. IEEE/ACM Trans. Netw. 11(4), 663–675 (2003)
14. Gouda, M.G., Schneider, M.: Stabilization of maximal metric trees. In: Arora, A. (ed.) WSS, pp. 10–17. IEEE Computer Society, Los Alamitos (1999)
15. Huang, S.T., Chen, N.S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. 41(2), 109–117 (1992)
16. Dubois, S., Masuzawa, T., Tixeuil, S.: Maximum Metric Spanning Tree made Byzantine Tolerant. Research report (2011), `http://hal.inria.fr/inria-00589234/en/`
17. Dubois, S., Masuzawa, T., Tixeuil, S.: Self-Stabilization, Byzantine Containment, and Maximizable Metrics: Necessary Conditions. Research report (2011), `http://hal.inria.fr/inria-00577062/en`

Performing Dynamically Injected Tasks on Processes Prone to Crashes and Restarts*

Chryssis Georgiou[1] and Dariusz R. Kowalski[2]

[1] Department of Computer Science, University of Cyprus, Cyprus
chryssis@cs.ucy.ac.cy

[2] Department of Computer Science, University of Liverpool, UK
D.Kowalski@liverpool.ac.uk

Abstract. To identify the tradeoffs between efficiency and fault-tolerance in dynamic cooperative computing, we initiate the study of a task performing problem under dynamic processes' crashes/restarts and task injections. The system consists of n message-passing processes which, subject to dynamic crashes and restarts, cooperate in performing independent tasks that are continuously and dynamically injected to the system. The task specifications are not known a priori to the processes. This problem abstracts todays Internet-based computations, such as Grid computing and cloud services, where tasks are generated dynamically and different tasks may be known to different processes. We measure performance in terms of the number of pending tasks, and as such it can be directly compared with the optimum number obtained under the same crash-restart-injection pattern by the best off-line algorithm. We propose several deterministic algorithmic solutions to the considered problem under different information models and correctness criteria, and we argue that their performance is close to the best possible offline solutions.

Keywords: Performing tasks, Dynamic task injection, Crashes and restarts, Competitive analysis, Distributed Algorithms.

1 Introduction

Motivation. One of the fundamental problems in distributed computing is to have a collection of processes to collaborate in performing large sets of tasks. For such distributed collaboration to be effective it must be designed to cope with dynamic perturbations that occur in the computation medium (e.g., processes or communication failures). For this purpose, a vast amount of research has been dedicated over the last two decades in developing fault-tolerant algorithmic solutions and frameworks for various versions of such cooperation problems (e.g., [9,12,16,17,21]) and in deploying distributed collaborative systems and applications (e.g., [2,11,19,20]).

* The work of the first author is supported by research funds of the University of Cyprus. The work of the second author is supported by the Engineering and Physical Sciences Research Council [grant numbers EP/G023018/1, EP/H018816/1].

In order to identify the tradeoffs between efficiency and fault-tolerance in distributed cooperative computing, much research was devoted in studying the abstract problem of using n processes to cooperatively perform m independent tasks in the presence of failures (see for example [10,16,18]). In this problem, known as *Do-All*, the number of tasks m is assumed to be fixed and known a priori to all processes. Although there are several applications in which the knowledge of tasks can be known a priori, in todays typical Internet-based computations, such as Grid computing (e.g., [11]), Cloud services (e.g., [2]), and master-worker computing (e.g., [19,20]), tasks are generated dynamically and different tasks may be known to different processes. As such computations are becoming the norm (and not the exception) there is a corresponding need to develop efficient and fault-tolerant algorithmic solutions that would also be able to cope with dynamic tasks injections.

Our Contributions. In this work, in an attempt to identify the tradeoffs between efficiency and fault-tolerance in dynamic cooperative computing, we initiate the study of a task performing problem in which n message-passing processes, subject to dynamic crashes and restarts, cooperate in performing independent tasks that are continuously and dynamically injected to the system. The computation is broken into synchronous rounds; unless otherwise stated, we assume that tasks are of unit-length, that is, it takes one round for a process to perform a task. An execution of an algorithm is specified under a crash-restart-injection pattern. Then, the efficiency of an algorithm is measured in terms of the *maximum number of pending tasks* at the beginning of a round of an execution, taken over all rounds and all executions. This enables us to view the problem as an online problem and pursue competitive analysis [23], i.e., compare the efficiency of a given algorithm with the efficiency of the best offline algorithm that knows a priori the crash-restart-injection patterns.

Task performance guarantees: The first property we consider, which constitutes the basic *correctness* property, requires that no task is lost, that is, a task is either performed or the information of the task remains in the system. The second and stronger property, which we call *fairness*, requires that all tasks injected in the system are eventually performed.

Our approach: We deploy an *incremental* approach in studying the problem. We first assume that there is a centralized authority, called *central scheduler*, that at the beginning of each round informs the processes (that are currently operational) about the tasks that are still pending to be performed, including any new tasks injected in this round. The reason to begin with this assumption is two-fold: (a) The fact that processes have consistent information on the number of pending tasks enables us to focus on identifying the inherent limitations of the problem under processes failures/restarts and dynamic injection of tasks without having to implement information sharing amongst processes. The algorithmic solutions developed under this *information* model are used as building blocks in versions of the problem that deploy weaker information models. Furthermore, lower bound results developed in this information model are also valid for weaker information models. (b) Studying the problem under this assumption has its own independent interest, as the central scheduler can be viewed as an abstraction of a monitor used for monitoring the computation progress and providing feedback to the computing elements. For example it could be viewed as a *master server* in Master-Worker Internet-based computations such as SETI [19] or Pregel [20], or as a *resource broker/scheduler* in Computational Grids such as EGEE [11].

We then limit the information provided to the processes. We consider a weaker centralized authority, called *central injector*, which informs processes, at the beginning of each round, only about the tasks injected in this round and information about which tasks have been performed only in the previous round. We show how to transform solutions for the task performing problem under the model of central scheduler into solutions for the problem under the model of central injector with the expense of sending a quadratic number of messages in every round. It also occurs that a quadratic number of messages must be sent in some rounds by any correct distributed solution for the considered problem in the model of central injector.

With the gained knowledge and understanding, we then show how processes can obtain common knowledge on the set of pending tasks without the use of a centralized authority. We now assume the existence of a *local injector* that injects tasks to processes without giving them any global information (for example, each process may be injected tasks that no other process in the system has been injected, or only a subset of processes may be injected the same task). The injector can be viewed, for example, as a local daemon of a distributed application that provides local information to the process that is running on. We show that solutions to this more general setting come with minimal cost to the competitiveness, provided that *reliable multicast* [7] is available.

Our results: We now summarize our results — all of them concern deterministic solutions.

(a) Solutions guaranteeing correctness: For the model of central scheduler, we show a lower bound of $\mathrm{OPT} + n/3$ on the pending-tasks competitiveness of any deterministic algorithm, even for algorithms that make use of messages and are designed for restricted forms of crash-restarts patterns. We claim that this lower bound result is valid in all other settings we consider. We then develop the near-optimal deterministic algorithm AlgCS that does not make any use of message exchange amongst processes and achieves $\mathrm{OPT} + 2n$ pending-tasks competitiveness. Using a *generic transformation* we obtain algorithm AlgCI for the model with central injector with the same competitiveness as algorithm AlgCS. Algorithm AlgCI has processes sending messages to each other in every round. Finally, we develop algorithm AlgLI for the model with local injector and we show that it achieves $\mathrm{OPT} + 3n$ pending-tasks competitiveness, under the assumption of reliable multicast. These results are presented in Sect. 3.

(b) Solutions guaranteeing fairness: The issue of fairness is far more complex than correctness; we show that it is necessary and sufficient to assume that when a process restarts it does not fail again in the next at least two consecutive rounds; under this restriction, called 2-*survivability*, we develop fair determnistic

algorithms AlgCSF, AlgCIF, and AlgLIF in the three considered information models and show that they "suffer" an additional additive surplus of n to their competitiveness, comparing to the algorithms that guarantee only correctness. An interesting observation is that fairness can only be guaranteed in infinite executions, otherwise competitive solutions are not possible, c.f. Sect. 4.
(c) Bounding communication: We show that in the models of central and local injector, if processes do not send messages to all other processes, then correctness (and thus also fairness) cannot be guaranteed, unless stronger restrictions are imposed on the crash-restart patterns. This result is detailed in Sect. 5.1.
(d) Non-unit-length tasks: For the above results we assumed that tasks are of unit-length, that is, they require one round to be performed by some process. The situation is even more complex when tasks may not be of unit-length. For the model of central scheduler, we show that if tasks have uniform length $d \geq 1$, that is, each task requires d consecutive rounds to be performed by a process, then a variation of algorithm AlgCS achieves $\text{OPT} + 3n$ pending-tasks competitiveness, under the correctness requirement. We conjecture that similar techniques can be applied to obtain competitive algorithms in the other information models and under the fairness requirement. Then we show that bounded competitiveness is not possible if tasks have different lengths, even under slightly restricted adversarial patterns. These results are given in Sect. 5.2.

The negative results of (c) and (d) give rise to interesting research questions and yield interesting future research directions. These are discussed in Sect. 6. *Omitted details and proofs can be found in [24].*

Related Work. The *Do-All* problem has been studied in several models of computation, including message-passing (e.g., [10,16]), shared-memory (e.g., [18]), partitionable networks (e.g., [15]), in the absence of communication (e.g., [22]) and under various assumptions on synchrony/asynchrony and failures. As already mentioned, the underlying assumption is that the number of tasks m is *fixed, bounded and known* a priori (as well as the task specifications) by all processes. The *Do-All* problem is considered solved when all tasks are performed, provided that at least one process remains operational in the entire computation (this can be viewed as a simplified version of our *fairness* property). The efficiency of *Do-All* algorithms is measured either as the total number of tasks performed – *work complexity* [10] or as the total number of *available processes steps* [18]. Georgiou et al. [14] considered an iterated version of the problem, where *waves* of m tasks must be performed, one after the other. All task waves are assumed to be known a priori by the processes. Clearly the problem we consider in this work is more general (and harder), as tasks do not come in waves, are not known a priori, and their number might not be bounded. Furthermore, we consider processes crashes and restarts, as opposed to the work in [14] that considers only processes crashes. Chlebus et al. in [7] considered the *Do-All* problem in the synchronous message-passing model with processes crashes and restarts. In order to obtain a solution for the problem in this setting, they made two modeling assumptions: (a) Reliable multicast: if a process fails while mutlicasting a message, then either all (non-faulty) targeted processes receive the message, or none does, and (b) There is at least one process alive for $k > 1$ consecutive rounds of the computation. In the present paper, as already mentioned, we also require reliable multicast in the model with local injector, and as we discuss in later sections, to guarantee fairness we require a similar restriction on the process living period. Finally, in [15] an online version of the *Do-All* problem is considered where the network topology changes dynamically and processes form disjoint communication groups. In this setting the efficiency (work complexity) of a randomized *Do-All* algorithm is compared with the efficiency of an offline algorithm that is aware a priori of the changes in the network topology. Again, the number of tasks is fixed, bounded and known a priori to all processes.

The notion of competitiveness was introduced by Sleator and Tarjan [23] and it was extended for distributed algorithms in a sequence of papers by Bartal et al. [6], Awerbuch et al. [5], and Ajtai et al. [1]. Several distributed computing problems have been modeled as online problems and their competitiveness was studied. Examples include distributed data management (e.g., [6]), distributed job scheduling (e.g., [5]), distributed collect (e.g., [8]), and set-packing (e.g., [12]).

In a sequence of papers (e.g., [9,21]) a scheduling theory is being developed for scheduling computations having intertask dependencies for Internet-based computing. The task dependencies are represented as directed acyclic tasks and the theory has been extending the families of DAGs that optimal schedules can be developed. This line of work mainly focuses on exploiting the properties of DAGs in order to develop schedules. Our work, although it considers independent tasks, focuses instead, on the development of distributed fault-tolerant task performing algorithms and exploring the limitations of online distributed collaboration.

2 Model

Distributed Setting. We consider a distributed system consisting of n synchronous, fault-prone, message-passing processes, with unique ids from the set $[n] = \{1, 2, \ldots, n\}$. We assume that processes have access to a global clock. We further assume a fully connected underlying communication medium (that is, each process can directly communicate with every other process) where messages are not lost or corrupted in transit.

Rounds. For a simplicity of algorithm design and analysis, we assume that a single round is split into four consecutive steps: (a) Receiving step, in which a process receives messages sent to it in the previous round; (b) Task injection step, in which new tasks are injected to processes, if any; (c) Local computation step, in which a process performs local computation, including execution of at most one task; and (d) Sending step, in which a process sends messages to other processes as scheduled in the local computation part.

Tasks. Each task specification τ is a tuple (id, ρ, $code$), where $\tau.id$ is a positive integer that uniquely identifies the task in the system, $\tau.\rho$ corresponds to the round number that the task was first injected to the system to some process (or set of processes), and $\tau.code$ corresponds to the computation that needs to occur so that the task is considered completed (that is, the computational part of the task specification that is actually performed). *Unless otherwise stated, c.f., Sections 5.2 and 6*, we assume that it takes one round for each task to be performed, and it can be performed by any process which is alive and knows the task specification.

Tasks are assumed to be similar, independent and idempotent. By *similarity* we mean that the task computations on any process consume equal or comparable local resources. By *independence* we mean that the completion of any task does not affect any other task, and any task can be performed concurrently with any other task. By *idempotence* we mean that each task can be performed one or more times to produce the same final result. Several applications involving tasks with such properties are discussed in [16]. Finally, we assume that task specifications are of polynomial size in n.

Adversary. We assume an adaptive and omniscient adversary that can cause crashes, restarts and task injections. We define an adversarial pattern $\mathcal{A}$ as a collection of crash, restart and injection events caused by the adversary. A $crash(r, i)$ event specifies that process i is crashed in round r. A $restart(r, i)$ event specifies that process i is restarted in round r; it is understood that no $restart(r, i)$ event can take place if there is no preceding $crash(r', i)$ event such that $r' < r$ (unless i is first entering the computation). Finally an $inject(r, i, \tau)$ event specifies that process i is injected the task specification τ in round r.

We say that a process i is *alive* in a round r if the process is operational at the beginning of the round and does not fail by the end of the round (a process that restarts at a beginning of a round and does not fail by the end of the round is also considered alive in that round). We assume that when the adversary injects tasks in a given round, it injects a finite number of tasks.

Restarts of processes: We assume that a restarted process has knowledge of only the algorithm being executed and the ids of the other system processes (but no information on which processes are currently alive). Algorithmically speaking, once a process restarts, it waits to receive messages or to be injected tasks. Then it knows that a new round has begun and hence it can smoothly start actively participating in the algorithm. For the ease of analysis and better clarity of result exposition we simply assume that processes are restarted at the beginning of a round – but processes could fail at any point during a round. We also assume that a process that restarts in the beginning of round r receives the messages sent to it (if any) at the end of round $r - 1$.

Admissibility: We say that an adversarial pattern is *admissible*, if

(a) in every round there is at least one alive process; in case of finite executions, all processes alive in the last round are crashed right after this round (in other words, a finite execution of an algorithm ends when all processes are crashed); and
(b) a task τ that is injected in a given round is injected to at least one alive process in that round; that is, the adversary gives some window of opportunity for task τ to either be performed in that round or other processes to be informed about this task.

Condition (a) is required to guarantee some progress in the computation. To motivate condition (b), consider the situation where a process in a given round is injected a task τ (and this is the only process being injected task τ) and then the process immediately crashes. No matter of the scheduling policy or communication strategy used, task τ cannot be performed by any algorithm; with condition (b) we exclude the consideration of such uninteresting cases; these tasks are not taken into consideration, neither for correctness, nor for performance issues. From this point onwards we only consider admissible adversarial patterns.

Restricted Classes of Adversaries. As we show later, some desired properties of task performing algorithms, such as fairness, may not be possible to achieve in general executions under any admissible adversarial pattern. In such cases, we also consider a natural property that restricts the power of adversary, called *t-survivability*: Every awaken process must stay alive for at least t consecutive rounds, where $t \geq 1$ is an integer.

Information Models. In regards to the distribution of injected tasks, as discussed in Section 1, we study three settings:

(i) *central scheduler*: in the beginning of each round it provides all operational processes with the current set of unperformed tasks' specifications;
(ii) *central injector*: in the beginning of each round it provides all operational processes with specifications of all newly injected tasks, and also confirmation of tasks being performed in the previous round, i.e., for round r, it informs all operational processes of the tasks injected in this round and the tasks that have been successfully performed in round $r-1$;
(iii) *local injector*: in the beginning of each round it provides each operational process with specifications of tasks injected into this process; a task specification may be injected to many processes in the same round.

Correctness and Fairness. We consider two important properties of an algorithm: correctness and fairness.

Correctness of an algorithm: An algorithm is *correct* if for any execution of the algorithm under an *admissible adversarial pattern*, for any injected task and any round following the injection time, there is a process alive in this round that stores the task specification, unless the task has been already performed. Observe that this property does not guarantee eventual performance of a task.

Fairness of an algorithm: We call an *infinite* execution of an algorithm under an adversarial pattern *fair execution* if each task injected during the execution is eventually performed. We say that an algorithm is a *fair algorithm* if every infinite execution of this algorithm is fair; note that the adversary can form *finite* executions in which not all tasks can be performed, not even by the offline algorithm. In other words, this property requires correctness, plus the guarantee that each task is eventually performed in any infinite execution of an algorithm. Observe that the greedy offline algorithm described above is fair.

Efficiency Measures. *Per round pending-tasks complexity*: Let P_r denote the total number of pending tasks at the beginning of round r, where by *pending task* we understand a task which has been already injected to some process (or a set of processes) but not yet performed by any process.[1] Then the per round pending-tasks complexity is defined as the maximum P_r over all rounds (supremum in case of infinite computations).

In case of competitive analysis, we say that the competitive pending-tasks complexity is $f(\mathrm{OPT}, n)$, for some function f, if and only if for every adversarial pattern $\mathcal{A}$ and round r the number of pending task in round r of the execution of the algorithm against adversarial pattern $\mathcal{A}$ is at most $f(\mathrm{OPT}(\mathcal{A}, r), n)$, where $\mathrm{OPT}(\mathcal{A}, r)$ is the minimum number of pending tasks achieved by an off-line algorithm, knowing $\mathcal{A}$, in round r of its execution under the adversarial pattern $\mathcal{A}$. In the classical competitiveness methodology function f needs to be linear with respect to the first coordinate, however as we will show, sometimes more accurate functions can be produced for the problem of distributed task performance.

Observe that the above definition allows for the optimum complexity of two different rounds to be met by two different optimum algorithms. However a simple greedy algorithm scheduling different pending tasks (with largest possible pending time) to different alive processes at each round is optimal from the perspective of any admissible adversarial pattern $\mathcal{A}$ and any round r (recall that to specify the optimum algorithm we can use the knowledge of $\mathcal{A}$).

For the sake of more sensitive bounds on competitiveness of algorithms, we consider subclasses of adversarial paterns achieving the same worst-case performance in terms of the optimum solution. These classes are especially useful for establishing sensitive lower bounds. We say that an adversarial pattern $\mathcal{A}$ is (k, r)-*dense* if $\mathrm{OPT}(\mathcal{A}, r) = k$. A pattern $\mathcal{A}$ which is (k, r)-dense for some round r is called k-*dense*.

In Section 5.1 we also study the message complexity of solutions to the task performing problem. Specifically we consider *per-round message complexity*, defined as the maximum number of point-to-point messages sent in a single round of an execution of a given algorithm, over all executions and rounds.

3 Solutions Guaranteeing Correctness

In this section we study the problem focusing on developing solutions that guarantee correctness, but not necessarily fairness (this property is studied in Section 4). We consider unit-length tasks (non-unit-length tasks are discussed in Section 5.2). We impose no restriction on the number of messages that can be sent in a given round; for example processes could send a message to every other processes, in every round (the issue of restricted communication is studied in Section 5.1). The results of this section are obtained in the most general of considered settings: the upper bounds hold against any admissible adversary,

[1] If a task was performed by some process, but the adversary did not provide the possibility to this process to inform another process or a central authority (scheduler or injector) — e.g., the process is crashed as soon as it performs the task — then this task is not considered performed.

while the lower bounds hold even in the presence of a restricted adversary satisfying t-survivability, for any t.

3.1 Central Scheduler

We first show that all algorithms require a linear additive factor in their competitive pending-tasks complexity. This bound holds for all settings considered in this work, as the central scheduler is the most restrictive one.

Theorem 1. *Every algorithm has competitive pending-tasks complexity of at least $k + n/3$ against some k-dense adversarial pattern satisfying t-survivability, for every non-negative integers k, t.*

Next, we show that the following simple algorithm, specified for a process i and a round r, is near-optimal. Observe that the algorithm does not require sending messages between processes.

Algorithm AlgCS(i, r)

```
– Get set of pending task specifications from the scheduler.
– Rank the task specifications in incremental order, based on the task id ($\tau.id$ for a
  task specification $\tau$).
– Perform task with rank $i$ mod $n$.
```

Theorem 2. *Algorithm AlgCS achieves competitive pending-tasks complexity of at most $OPT + 2n$ against any admissible adversary.*

3.2 Central Injector

We now relax the information given to the processes in the beginning of every round by considering the weaker model of central injector. We first show how to *transform* an algorithm specified for the setting with central scheduler, call it *source algorithm*, into an algorithm specified for the setting with central injector, call it *target algorithm*. The transformation maintains all local variables used by the source algorithm and sends the same messages, but now additional local variables are used and messages may contain additional information, required by the processes in the target algorithm in order to obtain the same set of pending tasks (under the same adversarial pattern) as the one that the central scheduler provides by default to the processes in the source algorithm.

The main structure of a generic algorithm, call it GenCS, specified for the setting with central scheduler is as follows (for a process i and round r):

Source Algorithm GenCS(i, r)

```
– Get set $\mathcal{P}$ of pending task specifications from the scheduler.
  Receive messages by each process $j$ that sent a message in the previous round
  containing information $x_j$.
– Based on $\mathcal{P}$ and each received information $x_j$ deploy the scheduling policy $\mathcal{S}$ to
  perform a task.
– Send a message with information $x_i$ to all other processes.
```

We now present the target algorithm, call it GenCI, which is obtained when we deploy our transformation, call it *tranCStoCI*, to the source algorithm GenCS. The text in bold annotates the elements added from *tranCStoCI* (these elements essentially specify the transformation).

Target Algorithm GenCI(i, r)

- **Get set $\mathcal{N}$ of specifications of newly injected tasks and set $\mathcal{D}$ of tasks confirmed as done in round $r-1$, from the central injector.**
  Receive messages by each process j that sent a message in the previous round containing information x_j**, and $\mathcal{P}_j$. Let R** $= \{j$ **: received a message from** j **in this round**$\}$.
- **Update local set $\mathcal{P}_i$ of pending tasks as follows:** $\mathcal{P}_i = \bigcup_{j \in R \cup \{i\}} \mathcal{P}_j \cup \mathcal{N} \setminus \mathcal{D}$.
- Based on $\mathcal{P}_i$ and each received information x_j deploy the scheduling policy $\mathcal{S}$ to perform a task.
- Send a message with information x_i **and** $\mathcal{P}_i$ to all other processes.

It is evident that algorithm GenCI continues to maintain the variables of GenCS and sends the same messages as algorithm GenCS (but with more information). What remains to show is that the set of pending tasks used in the scheduling policy $\mathcal{S}$ in a given round is the same for both algorithms.

Lemma 1. *For any given round r, the set of pending tasks used in the scheduling policy $\mathcal{S}$ is the same in the executions of algorithms GenCS and GenCI formed by the same adversarial pattern.*

Consider algorithm AlgCS of Section 3.1. This algorithm is a specialization of algorithm GenCS where the x_i's are *null* and the scheduling policy $\mathcal{S}$ is simply ranking the tasks in $\mathcal{P}$ in incremental order (based on their ids) and having process i perform task with rank i mod n. Let **Algorithm AlgCI** be the algorithm resulting by applying the transformation *TranCStoCI* to algorithm AlgCS. Then, from Lemma 1 and Theorem 2 we get:

Theorem 3. *Algorithm AlgCI achieves competitive pending-tasks complexity of at most $OPT + 2n$ against any admissible adversary.*

In view of the lower bound in Theorem 1, algorithm AlgCI is near-optimal.

3.3 Local Injector

In this section we consider the local injector model. Consider algorithm AlgLI, specified below for a process i and a round r. In each round r, each process i maintains two sets, *new* and *old*. Set *new* contains all new tasks injected to this process in this round. Set *old* contains older tasks that the process knows they have been injected in the system (not necessarily to this process) but have not been confirmed as done. We show the following:

Theorem 4. *Algorithm AlgLI, assuming reliable multicast, is correct and near-optimal; more precisely, it achieves competitive pending-tasks complexity of at most $OPT + 3n$ against any adversary.*

Algorithm AlgLI(i, r)

- Get specifications of newly injected tasks from local scheduler and store them in set *new* (remove any older information from this set).
  Receive messages sent (if any) by other processes in round $r - 1$.
- Update set *old* based on received messages: the new set *old* is the union of all the received sets *old* and *new* minus the tasks that have been reported in the current round as done in the previous round.
- Perform a task based on the following scheduling policy: if set $old \neq \emptyset$ then rank tasks in *old* incrementally based on their ids and perform task with rank $i \mod |old|$. Otherwise, and if *new* is not empty, then rank the tasks in *new* incrementally based on their ids and perform task with the smallest rank.
- Send to all other processes sets *new*, *old* and the task id of the performed task.

4 Solutions Guaranteeing Fairness

We now turn our attention to the much challenging problem of guaranteeing fairness. Recall from Section 2 that for fairness we consider only infinite executions and for such executions there is always a fair (offline) algorithm.

4.1 Central Scheduler

We first demonstrate that the issue of fairness is much more involved than correctness. Consider the following simple fair algorithm LIS: each process performs the Longest-In-System task, and in case of a tie it chooses the one with the smallest task id.

Fact 5. *Algorithm LIS has unbounded pending-tasks competitiveness under any adversary, even for the restricted one satisfying t-survivability, for any $t \geq 1$.*

The above shows that a fair algorithm not only needs to have some provision in eventually performing a specific task but it also needs to guarantee progress when a large number of tasks is pending. Furthermore, we show that admissibility alone is not enough to guarantee both fairness and bounded competitiveness.

Theorem 6. *For any fair algorithm and any integer $y > 0$, there is a round r and an admissible, adversarial pattern $\mathcal{A}$ such that the algorithm has more than $y \cdot (OPT(\mathcal{A}, r) + 1)$ pending tasks at the end of round r.*

Note that Theorem 6 implies that the algorithms presented in Section 3 are not fair. Therefore, in order to achieve both fairness and competitiveness, one needs to consider some restrictions to the adversary. It can be easily verified that the impossibility statement in Theorem 6 holds even if 1-survivability is assumed. As it turns out, it is enough to assume 2-survivability to be able to obtain fair and competitive algorithms.

Consider algorithm AlgCSF specified below for process i and round r. Each process i maintains a variable *age* that counts the number of rounds that i has been alive since it last restarted. A restarted process has $age = 0$, and it increments it by one at the end of each local computation part. For simplicity, we

Algorithm AlgCSF(i,r)

```
– Get pending tasks from central scheduler and messages sent (if any) in round $r-1$.
– Rank pending tasks lexicographically: first based on their pending period (older
  tasks have smaller rank) and then based on their task ids (incremental order).
– Based on received messages, construct set $ASure$ by including all processes $j$ with
  $age_r(j) = 1$. If $age = 1$, then $i$ includes itself in the set.
– If the number of pending tasks is larger than $2n$ then
    • If $ASure \neq \emptyset$ then
        ∗ If $age \neq 1$ then perform task with rank $n+i$.
        ∗ Else rank processes in $ASure$ based on their ids and perform task with
          rank $rank(i)_{ASure}$ (i.e., $i$th task in ranked set $ASure$).
    • Else [$ASure = \emptyset$]
        ∗ If $age \neq 0$ then construct set $Recved$ by including all processes from
          whom a message was received at the beginning of the round. Process $i$
          includes itself in this set. Then rank processes in set $Recved$ lexicograph-
          ically, first based on their age and then based on id (increasing order). If
          $rank(i)_{Recved} = 1$ then perform task with rank 1, otherwise perform task
          with rank $i+1$.
        ∗ Else perform task with rank $i+1$.
  Else perform task with rank 1.
– Set $age = age + 1$.
– Send $age$ to all other processes as the value of variable $age_{r+1}(i)$.
```

say that in round r process is in age x if it was alive for the whole x rounds, i.e., its *age* is x in the beginning of the round. Processes exchange these variables, so, for reference reasons, we will be denoting by $age_r(j)$ the age that process i knows that j has in round r (in other words, this is the age j reports to i at the end of round r).

We begin to show that algorithm AlgCSF is fair, under the assumption of 2-survivability.

Lemma 2. *If in a given round r, τ_{old} is the oldest pending task in the system (has rank 1) and there is at least one process with $age_r = 1$, then τ_{old} is performed by the end of round r.*

Observe that if in round r there is no process with age 1 but there is at least one with age 0, then even if τ_{old} is not performed in round r, by Lemma 2 it will be performed in round $r+1$. Hence it remains to show the following.

Lemma 3. *If in round r all alive processes are of age > 1 ($ASure = \emptyset$) and τ_{old} is the oldest task in the system, then τ_{old} will be performed by round $r + 2n$ at the latest.*

Lemmas 2 and 3 yield fairness of algorithm AlgCSF:

Theorem 7. *Algorithm AlgCSF is a fair algorithm under any 2-survivability adversarial pattern.*

It remains to show the competitiveness of algorithm AlgCSF, and this we show against *any* admissible adversarial pattern (unlike fairness, which is guaranteed if the pattern satisfies 2-survivability).

Theorem 8. *Algorithm AlgCSF achieves competitive pending-tasks complexity of at most $OPT + 3n$ against any admissible adversary.*

It can be seen that the lower bound stated in Theorem 1 is also valid for fair algorithms run against any admissible adversary, even the one restricted by t-survivability. Hence, we may conclude that algorithm AlgCSF, as well as fair algorithms AlgCIF and AlgLIF of competitiveness $\mathrm{OPT} + O(n)$ developed in the subsequent Sections 4.2 and 4.3 for central and local injectors, are near-optimal.

4.2 Central Injector

Recall transformation $TranCStoCI$ from Section 3.2. It is easy to see that algorithm AlgCSF is a specialization of the generic algorithm GenCS: information x_i is the age of process i. The remaining specification of algorithm AlgCSF (along with the required data structures) is essentially the specification of the scheduling policy $\mathcal{S}$ in the setting with central scheduler. Now, let **Algorithm AlgCIF** be the algorithm resulting by applying the transformation $TranCStoCI$ to algorithm AlgCSF (it is essentially algorithm GenCI appended with the scheduling policy of algorithm AlgCSF). Then, from Lemma 1 and Theorems 7,8 we get:

Theorem 9. *Algorithm AlgCIF is a fair algorithm that achieves competitive pending-tasks complexity of at most $OPT + 3n$ under any 2-survivability adversary.*

4.3 Local Injector

We now consider algorithm AlgLIF. This algorithm combines the mechanism deployed by algorithm AlgLI for propagating newly injected tasks with a round of delay and the scheduling policy of algorithm AlgCSF to guarantee fairness. Reliable multicast is again assumed for assuring that processes maintain consistent sets of pending tasks. A full description of algorithm AlgLIF is given in [24] (it is essentially a combination of the descriptions of the two above-mentioned algorithms). Its competitiveness is the same as the competitiveness of AlgCSF plus an additive factor n coming from the one-round delay of the propagation of newly injected tasks. Specifically we have that:

Theorem 10. *Algorithm AlgLIF, assuming reliable multicast, is a fair algorithm that achieves competitive pending-tasks complexity of at most $OPT + 4n$ against any 2-survivability adversary.*

5 Extensions and Limitations

In this section we consider the impact of restricted communication and non-unit-length tasks on the competitiveness of the problem of performing tasks under dynamic crashes-restarts-injections patterns.

5.1 Solutions under Restricted Communication

In view of Theorems 1 and 2, we argue that exchanging messages between processes does not help much in the setting with central scheduler, in the sense that in the best case it could slightly increase only the constant in front of the additive linear part of the formula on the number of pending tasks. In this section we study the problem of how exchanging messages may influence the correctness of solutions in more restricted settings of injectors. In particular, we show that $\Omega(n^2)$ per-round message complexity is inevitable in order to achieve correctness even in the presence of central injector. On the other hand, recall that $O(n^2)$ per-round message complexity is enough to achieve near-optimal solution in the presence of central injector: algorithm AlgCI from Section 3.2 achieves near-optimal competitiveness of at most $\mathrm{OPT} + 2n$ in this setting, c.f., Theorem 3.

Theorem 11. *For any algorithm and any $t \geq 1$, there is an adversarial pattern satisfying t-survivability such that the execution of the algorithm under this pattern results in $\Omega(n^2)$ per-round message complexity, even in the model with central injector.*

5.2 Non-unit-Length Tasks

We now turn our attention to tasks that are not necessarily of unit-length, that is, they might take longer than a round to complete. We consider a persistent setting, in which once a process commits in performing a certain task of length x, it will do so for x consecutive rounds, until the task is performed. If the process is crashed before the completion of all x rounds, then the task is not completed. We assume that processes cannot share information of partially completed tasks; the task performance is an atomic operation. In view of these assumptions, the number of pending tasks remains a sensible performance metric.

First, we consider tasks of the same length $d \geq 1$, i.e., each task takes d rounds to be performed. Consider a variation of algorithm AlgCS of Section 3.1 that uses the same scheduling policy, but once a process chooses a task to perform, it spends d consecutive rounds in doing so; call this AlgCS_d. We show the following:

Theorem 12. *Algorithm $AlgCS_d$, for uniform tasks of length d, achieves competitive pending-task complexity of at most $OPT + 3n$ under any admissible adversarial pattern, in the setting with central scheduler.*

We conjecture that similar techniques would lead to near-optimal analysis for the other algorithms developed in this paper, in the context of the remaining two models of central and local injectors, and under the fairness requirement.

We now consider the case where tasks could be of *different* lengths. It follows that bounded competitiveness is not possible, even under restricted adversarial patterns, and even in the model with central scheduler.

Theorem 13. *For any algorithm, any number $n \geq 2$ of processes, any $t \geq 1$ and any upper bound $d \geq 3$ on the lengths of tasks, there is an adversarial pattern satisfying t-survivability such that the execution of the algorithm under this pattern results in unbounded competitiveness with respect to the pending task complexity, even in the model with central scheduler.*

6 Future Directions

Several research directions emanate from this work. An intriguing question is whether the assumption of reliable multicast, made in the setting with local injector, can be removed or replaced by a weaker but still natural constraint. We conjecture that t-survivability, for a suitable constant t, could be a good candidate for such replacement. In view of Theorem 11, it is challenging to find a natural restriction on the adversary such that both efficient performance and *subquadratic communication* would be achieved in the settings with injectors. For this purpose a version of the continuous gossip protocol developed in [13] could be possibly used. In view of Theorem 13, it would be worth checking if randomization would help (i.e., analyzing randomized algorithms under oblivious adversaries), or whether a smoothed or average-case analysis might result in bounded competitiveness for tasks of different lengths.

Another interesting challenge is to generalize the considered task specifications to dependent tasks. Other challenging modeling extensions could involve replacing the fairness property by a more "sensitive" task latency measure, and considering energy consumption issues.

References

1. Ajtai, M., Aspnes, J., Dwork, C., Waarts, O.: A theory of competitive analysis for distributed algorithms. In: Proc. of FOCS 1994, pp. 401–411 (1994)
2. Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2`
3. Attiya, H., Fouren, A.: Polynomial and adaptive long-lived ($2k$ - 1)-renaming. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 149–163. Springer, Heidelberg (2000)
4. Attiya, H., Fouren, A., Gafni, E.: An adaptive collect algorithm with applications. Distributed Computing 15(2), 87–96 (2002)
5. Awerbuch, B., Kutten, S., Peleg, D.: Competitive distributed job scheduling. In: Proc. of STOC 1992, pp. 571–580 (1992)
6. Bartal, Y., Fiat, A., Rabani, Y.: Competitive algorithms for distributed data management. In: Proc. of STOC 1992, pp. 39–50 (1992)
7. Chlebus, B., De-Prisco, R., Shvartsman, A.A.: Performing tasks on restartable message-passing processors. Distributed Computing 14(1), 49–64 (2001)
8. Chlebus, B.S., Kowalski, D.R., Shvartsman, A.A.: Collective asynchronous reading with polylogarithmic worst-case overhead. In: Proc. of STOC 2004, pp. 321–330 (2004)
9. Cordasco, G., Malewicz, G., Rosenberg, A.: Extending IC-Scheduling via the sweep algorithm. J. of Parallel and Distributed Computing 70(3), 201–211 (2010)
10. Dwork, C., Halpern, J., Waarts, O.: Performing work efficiently in the presence of faults. SIAM Journal on Computing 27(5), 1457–1491 (1998)
11. Enabling Grids for E-sciencE (EGEE), `http://www.eu-egee.org`
12. Emek, Y., Halldorsson, M.M., Mansour, Y., Patt-Shamir, B., Radhakrishnan, J., Rawitz, D.: Online set packing and competitive scheduling of multi-part tasks. In: Proc. of PODC 2010, pp. 440–449 (2010)
13. Georgiou, C., Gilbert, S., Kowalski, D.R.: Meeting the deadline: on the complexity of fault-tolerant continuous gossip. In: Proc. of PODC 2010, pp. 247–256 (2010)

14. Georgiou, C., Russell, A., Shvartsman, A.A.: The complexity of synchronous iterative Do-All with crashes. Distributed Computing 17, 47–63 (2004)
15. Georgiou, C., Russell, A., Shvartsman, A.A.: Work-competitive scheduling for cooperative computing with dynamic groups. SIAM J. on Comp. 34(4), 848–862 (2005)
16. Georgiou, C., Shvartsman, A.A.: Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity. Springer, Heidelberg (2008)
17. Hui, L., Huashan, Y., Xiaoming, L.: A Lightweight Execution Framework for Massive Independent Tasks. In: Proc. of MTAGS 2008 (2008)
18. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers, Dordrecht (1997)
19. Korpela, E., Werthimer, D., Anderson, D., Cobb, J., Lebofsky, M.: SETI@home: Massively distributed computing for SETI. Comp. in Sc. & Eng. 3(1), 78–83 (2001)
20. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proc. of SIGMOD 2010, pp. 135–145 (2010)
21. Malewicz, G., Rosenberg, A., Yurkewych, M.: Toward a theory for scheduling dags in Internet-based computing. IEEE Trans. on Computers 55(6), 757–768 (2006)
22. Malewicz, G., Russell, A., Shvartsman, A.A.: Distributed scheduling for disconnected cooperation. Distributed Computing 18(6), 409–420 (2006)
23. Sleator, D., Tarjan, R.: Amortized efficiency of list update and paging rules. Communications of the ACM 28(2), 202–208 (1985)
24. Tech. Report of this work, http://www.cs.ucy.ac.cy/~chryssis/disc11-TR.pdf

Leakage-Resilient Coin Tossing

Elette Boyle[1,⋆], Shafi Goldwasser[1,2,⋆⋆], and Yael Tauman Kalai[3]

[1] MIT, Cambridge, MA, USA
eboyle@mit.edu
[2] Weizmann Institute of Science, Rehovot, Israel
[3] Micorosft Research New England, Cambridge, MA, USA

Abstract. The ability to collectively toss a common coin among n parties in the presence of faults is an important primitive in the arsenal of randomized distributed protocols. In the case of dishonest majority, it was shown to be impossible to achieve less than $\frac{1}{r}$ bias in $O(r)$ rounds (Cleve STOC '86). In the case of honest majority, in contrast, unconditionally secure $O(1)$-round protocols for generating common *unbiased* coins follow from general completeness theorems on multi-party secure protocols in the secure channels model (e.g., BGW, CCD STOC '88).

However, in the $O(1)$-round protocols with honest majority, parties generate and hold secret values which are assumed to be *perfectly hidden* from malicious parties: an assumption which is crucial to proving the resulting common coin is unbiased. This assumption unfortunately does not seem to hold in practice, as attackers can launch side-channel attacks on the local state of honest parties and leak information on their secrets.

In this work, we present an $O(1)$-round protocol for collectively generating an unbiased common coin, in the presence of leakage on the local state of the honest parties. We tolerate $t \leq (\frac{1}{3} - \epsilon)n$ computationally-unbounded Byzantine faults and in addition a $\Omega(1)$-fraction leakage on each (honest) party's secret state. Our results hold in the memory leakage model (of Akavia, Goldwasser, Vaikuntanathan '08) adapted to the distributed setting.

Additional contributions of our work are the tools we introduce to achieve the collective coin toss: a procedure for *disjoint committee election*, and *leakage-resilient verifiable secret sharing*.

1 Introduction

Randomization, and the ability to keep your local randomness and local state private, are fundamental ingredients at the disposal of fault tolerant distributed

⋆ Research supported by NDSEG graduate fellowship.
⋆⋆ This work was supported in part by Trustworthy Computing: NSF CCF-1018064. This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-11-2-0225. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

algorithms. This was realized originating with the work of Rabin [28], introducing the power of a shared global common coin to obtain a dramatic reduction in round complexity with respect to Ben-Or's asynchronous randomized consensus algorithm [3][1]; and continued to be utilized in many beautiful distributed algorithms to this day in various network models.

The assumption that a party's local state—including its local randomness and the values of its secret cryptographic keys—is perfectly hidden from an adversary is an assumption that has undergone much scrutiny in the past few years in the cryptographic community. This is in light of accumulating evidence which shows that in practice, physical measurements (so called side-channel attacks) can be made on honest parties' devices, resulting in leakage from their local state that can completely compromise the security of the cryptographic algorithm. Indeed, a considerable amount of effort in the cryptographic community is devoted today to develop new cryptographic schemes which are resistant to leakage (e.g., [20,25,13,2] and many more). Several models of leakage have been considered. The one most relevant to this work is that an adversary can adaptively choose *any* leakage functions, and receive the value of the leakage functions on the secret state of the device, as long as the total *amount* of leakage is bounded (shorter than the secret state) [2].

We propose to mirror this line of work in the regime of distributed fault tolerant algorithms. Namely, to address the question of how leakage from the local state of non-faulty parties affects the correctness of fault-tolerant distributed protocols. Here, in addition to the fact that some of the parties are faulty and fully compromised, the adversary who is coordinating the action of the faulty parties can obtain partial information in the form of leakage on the local state of each honest party. This may potentially enable the adversary to alter the course of the protocol's execution. We note that in this context, the coordinating adversary can adaptively choose the leakage function, depending on the history of communication it sees thus far.

In particular, in this paper we provide a fault-tolerant, leakage-resilient protocol for collective unbiased coin tossing among n parties.

The problem of collective coin tossing in a distributed setting has received a great deal of prior attention, starting with the work of Rabin [28] on distributed consensus. When there is no honest majority of parties, results from the two-party setting by [11] showed that a bias of $\frac{1}{r}$ must be incurred by any $O(r)$-round protocol (this was recently shown optimal in a work of Moran et al. [26]). Loosely speaking, the problem is that a dishonest party can do the following: At the last round, before sending his final message, he can compute the outcome, and abort if he does not favor this outcome, thus biasing the output. When there is an honest majority of parties, this attack can be prevented using *verifiable secret sharing* (VSS), a notion defined by Chor et al. [10]. Verifiable secret sharing allows each of the n parties to toss a coin locally and share it among the n parties. After all the local coins have been shared via a VSS, the parties reconstruct the

[1] Ben-Or's protocol does not require the local coin outcomes to ever remain private, only requiring they be random. Alas, the number of rounds is exponential.

values, and the output coin is set to be the xor of the local coins. The works of Ben-Or et al. [4] and Chaum et al. [9] on secure multi-party computation show how to achieve VSS—and thus how to construct an unbiased coin tossing protocol—in expected $O(1)$ rounds. These results ([4,9]) hold unconditionally in the synchronous network model with less than a third Byzantine faults, assuming perfectly secure channels of communication between pairs of users and the use of a broadcast channel.

However, each of these protocols require the parties to generate and hold secret values, and security is guaranteed only under the assumption that these secrets are *completely* hidden from the adversarial view. It is easy to check that correctness breaks down if the adversary obtains some partial information about these secrets. This is the starting point of our work.

1.1 Our Contributions

1.1.1 Leakage-Resilient Coin Tossing

We construct a leakage-resilient collective coin-tossing protocol in synchronous point-to-point networks with a broadcast channel and physically secure communication channels between pairs of parties.

We allow up to one third colluding, statically corrupted malicious parties. Namely, a computationally unbounded rushing adversary can a priori choose parties to corrupt; during the protocol, he sees the internal state of all corrupted parties and can set the messages of these parties at any round, as a function of all honest parties' messages up to (and including) this round. In addition, the adversary can make *leakage queries* at every round in the form of specifying a party and a leakage function, and obtains the result of his leakage functions applied to the internal state of the corresponding honest parties.

We allow the adversary to leak *arbitrary* functions of parties' secret states, as long as the total number of bits leaked from each party is at most some (pre-specified) λ fraction of its entire secret state.[2] Each leakage query is applied to the secret state of a single party. Since participants of a distributed protocol typically run on different physical hardware (and reside in different locations), we believe that it is reasonable to assume each leakage query modeling a physical measurement reveals information about each party's execution separately. To maximize generality within this setting, we allow the leakage queries on different parties' secret states to be interleaved (i.e., leak from party i, then from party j, and then again from party i), and the choice of leakage queries to be adaptively selected as a function of prior leakage. We remark that this distributed leakage model is similar to a model proposed by Akavia et al. [1] in their work on public-key encryption in which the secret key of the decryption algorithm is distributed among two parties.

We call a n-party distributed protocol (t, λ)-*leakage-resilient* if it achieves its desired functionality in the presence of an adversary who can control up to t

[2] Our methods extend to also tolerate the Naor and Segev [27] leakage model which allows leakage functions which are not necessarily shrinking but leave the internal local state with enough min-entropy.

parties and can leak up to a λ fraction of the internal secret state of each honest party (as above). We can now state our main theorem.

Theorem (Coin Toss). For any constants $\delta, \epsilon > 0$, any $\lambda \leq \frac{\delta(1-\epsilon)}{10+6\delta}$, any $n \geq (3+\delta)t$, and any m, there exists a (t, λ)-leakage-resilient n-party distributed protocol that outputs a value $v \in \{0,1\}^m$, and terminates in $O(1)$ rounds satisfying:

– **Agreement:** At the conclusion of the protocol, each party outputs a value $v_i \in \{0,1\}^m$. For all honest parties P_i, P_j, it holds that $v_i = v_j$.
– **Randomness:** With all but negligible probability (in n), the distribution of the honest output value v is statistically close to uniform.

A few remarks are in order.

Fairness: We emphasize that our protocol achieves fairness, in that honest parties output a random string even if dishonest parties abort prematurely.

Strings versus Bits: We note that the output of our coin tossing protocol can be a long random string, as opposed to just a single bit. In the leak-free setting, this point is not worth emphasizing, since the coin-tossing protocol can be run in parallel to output as many bits as desired. However, in the leaky setting, if we run many protocols in parallel, leakage bounds may deteriorate quickly: if we run k protocols, where each protocol tolerates leakage rate λ, then in the resulting parallel execution, the leakage rate becomes only $\frac{\lambda}{k}$. Thus, to maintain leakage bounds we would need to run the protocol sequentially, resulting in many rounds of communication. Our protocol has the property that it can output as many bits as desired in constant rounds with constant leakage rate.

Weakening the Secure Channels Assumption: We assumed physically secure channels; however, our leakage model immediately implies we can tolerate leakage of information from these channels, as parties' messages are computed as a function of public information and their personal secret state. To remove the secure channels assumption altogether, we would need to send the messages between honest parties using encryption, which would necessitate a computational assumption supporting the strength of the encryption algorithm. Furthermore, one would have to consider whether leakage from the secret keys of the decryption algorithm and the randomness used by the encryption algorithm can be tolerated. A recent work of Bitansky et al. [5] suggests to send messages encrypted with non-committing encryption (introduced by Canetti et al. [8]), protocols in the secure channels model can be transformed into leakage-resilient secure protocols that do not assume secure channels.

Relation to Using Imperfect Random Sources in Distributed Computing: The question of achieving $O(1)$-round Byzantine Agreement and multi-party computation when parties do not have access to perfect local randomness, but rather to independent imperfect random sources such as min-entropy sources [19,22,21], seems strongly related to our work here. Indeed, one may naturally view a random secret with leakage as a secret a-priori drawn from a min-entropy source.

The crucial difference between these works and our own is that our leakage model allows the adversary to leak adaptively *during* the protocol, as opposed to non-adaptively before the protocol begins. More specifically, the approach taken in [19,22,21] is to first generate *truly* random strings from the weak random sources, and then to use these random strings in the underlying protocol execution. This approach will not work in our setting, since the adversary can simply choose to leak on the newly generated random strings. On the other hand, we note that the works of [19,22,21] consider randomness coming from an *arbitrary* min-entropy distribution, whereas our model considers perfect randomness that is being leaked so as to leave min-entropy in the distribution.

Coin Flipping versus Byzantine Agreement: Achieving a weak form of collective coin tossing was an important building block to construct Byzantine agreement protocols in many works, most notably in the work of Dwork et al. [12], and of Feldman and Micali [15]. Our schemes to construct collective coin tossing utilize broadcast channels as a primitive (which are equivalent to Byzantine agreement), and thus obviously cannot be used to construct Byzantine agreement. It is an interesting question for future research how to achieve coin tossing in the presence of leakage without assuming broadcast channels.

Using Coin Tossing to Force Honest Behavior: An important technique in multi-party protocols, initially proposed by Goldwasser and Micali [18], is to force parties to use the result of a common coin toss as their local randomness, to ensure parties do not rig their coins. In this case, the result of the coin toss will be known only to one party Alice, and yet all other parties will be able to verify (via, say, zero-knowledge protocols) that Alice is using the result of the collective coins in her computations. This idea was later used in the compiler of [17] from the n-party secure function evaluation protocol with honest-but-curious majority to one with malicious majority. Our coin tossing protocol can similarly be turned into one where only one party Alice knows the result but all other parties can verify (via, say, a leakage-resilient zero knowledge protocol [16]) that Alice is using the result of the collective coins in her computations.

1.1.2 Leakage-Resilient Verifiable Secret Sharing

One of the tools in our construction, which is of independent interest, is a new *leakage-resilient verifiable secret sharing scheme.* Verifiable secret sharing (VSS) extends Shamir's [29] secret sharing to ensure not only secrecy (i.e., corrupted parties do not gain information about the dealer's secret), but also unique reconstruction of a secret s' even if the dealer and/or a subset of parties are dishonest, where for an honest dealer, s' will be his original secret. *Weakly leakage-resilient (WLR)* VSS is a VSS scheme with the additional guarantee that given the view of any (t, λ) adversary who corrupts up to t parties *and leaks* λ-fraction of each honest party's secret state (including the dealer's), the secret still retains a constant fraction of its original entropy. We refer to this property as weak leakage resilience. We now state our second main theorem.

Theorem (WLR-VSS): Let $n = (3 + \delta)t$ for some constant $\delta > 0$. Then for any constants $\epsilon < 1$ and $\lambda \leq \frac{\delta(1-\epsilon)}{10+6\delta}$, there exists a (t, λ)-leakage resilient VSS protocol that runs in $O(1)$ rounds, with the following modified secrecy guarantee: If the dealer is honest during the sharing phase, then for any (t, λ) adversary $\mathcal{A}$, with high probability, given the view of $\mathcal{A}$ at the conclusion of the sharing phase, the secret s retains ϵ fraction of its original entropy.

WLR-VSS is sufficient for our coin tossing construction; however, in the full version [6] we also define and obtain a stronger version of leakage-resilient VSS, in which given the view of a leaking adversary, the secret s retains its *full* entropy. This stringent secrecy property rules out the possibility of standard VSS, since leakage from the dealer directly reveals information on s. We thus put forth a new notion and a construction of *oblivious* secret sharing, where a dealer shares a uniformly distributed secret whose value *he does not know*. We believe that this primitive can serve as a useful building block for constructing future leakage-resilient protocols, which anyway make use of VSS in this fashion (e.g., in [15] to achieve Byzantine Agreement).

1.1.3 Disjoint Committee Election

As a tool in our construction, we present a 1-round public-coin protocol for electing $\log^2 n$ *disjoint* "good" committees of size approximately $n^{1/2}$ from among n parties. This is achieved using a modified version of the Feige committee election protocol [14] run in parallel, where recurring parties are removed from all but the first committee in which they appear.

1.2 Overview of Our Solution

Let us first see why simple and known coin tossing protocols are not resilient to leakage. Consider the following well-known coin tossing protocol paradigm: First, each party P_i chooses a random value r_i and secret shares it to all other parties using a *verifiable secret sharing* (VSS) protocol. Then, all the parties reveal their shares and reconstruct $r_1, \ldots, r_n$. Finally, the parties output $\oplus r_i$. This protocol is not resilient to leakage for several reasons.

First, the reduction from coin tossing to VSS fails. For example, a malicious party P_j can simply leak from each party P_i the least significant bit of r_i, and then choose r_j such that the xor of these least significant bits is zero. Thus, the problem is that in the leaky setting, we cannot claim that the r_i's look random to the adversary. Instead, all we can claim is that they have high min-entropy. To address this first problem, the first idea is to use a *multi-source extractor* instead of the xor function. Namely, output $\mathsf{Ext}(r_1, \ldots, r_n)$, where Ext is an extractor that takes n independent sources and outputs a string that is statistically close to uniform. Note however, that we cannot use any such multi-source extractor, since some of the sources (i.e., some of the r_j's) may be chosen maliciously. Thus, what we need is a multi-source extractor that outputs a (statistically close to) uniform string, even if some of the sources are arbitrary, but independent of the "honest" sources. Indeed, such an extractor was constructed by Kamp, Rao, Vadhan and Zuckerman [23].

Secondly, VSS protocols by and large are not resilient to leakage. Consider a single VSS protocol execution in the above paradigm. If the adversary leaks λ-fraction from each share, the total number of bits leaked is too large (indeed, potentially larger than the size of the secret being shared), and we cannot even guarantee that the secret r_i has any min entropy. Thus, we cannot use any VSS scheme, but rather we need to use a *leakage-resilient* one, with the guarantee that even if λ-fraction of each share is leaked, the secret still has high min-entropy. Indeed, we construct such a weakly leakage-resilient (WLR) VSS in Section 4. We note that many distributed protocols use VSS protocols, which immediately make them susceptible to leakage. Thus, our leakage-resilient VSS scheme may be useful for other protocols as well.

Finally, two technical difficulties remain. In the above coin-tossing paradigm utilizing WLR-VSS, each party shares his random value with all other n parties, and thus each honest party holds information on *all* secret values r_i. Since the leakage is computed on a party's entire secret state, the adversary may learn information on the joint distribution of the r_i's. This creates a dependency issue: recall that the output of the multi-source extractor is only guaranteed to be random if the sources r_i are independent. Further, in this paradigm the secret state of each party will be quite large, consisting of n secret shares (one for each secret value r_i). This will yield poor leakage bounds, with leakage rate less than $\frac{1}{n}$, if we want to ensure no share of one particular secret can be entirely leaked.

We avoid these problems by ensuring that each of the n parties will never hold more than one secret share of the r_i's. To this end, we follow a two-step approach. The first step is a universe reduction idea similar to the one going back to Bracha [7]. Instead of having *all* parties generate and secret share random strings r_i, we elect a small committee $\mathcal{E}$ (of size approximately $\log^2 n$), and only the members of $\mathcal{E}$ choose a random string r_i which will be shared via WLR-VSS (and later used in the construction of the collective coin). We utilize Feige's protocol [14] to elect this committee, which guarantees with high probability that the fraction of faulty parties in $\mathcal{E}$ is the same as in the global network. The second idea is that members of this committee do not WLR-VSS the r_i they chose to all n parties, but rather to small secondary committees. Namely, for every party $i \in \mathcal{E}$, all parties elect a secondary subcommittee $\mathcal{E}_i$, and party P_i will WLR-VSS her random string r_i only to parties in $\mathcal{E}_i$. We need to ensure that all the secondary committees $\mathcal{E}_i$ are disjoint, to avoid the case where one party has many shares. One may be tempted to simply force these committees to be disjoint by eliminating members that appear in previous committees. Indeed, we follow this approach. However, care must be taken when eliminating parties, since we may eliminate too many honest parties, and remain with dishonest majority. In Proposition 5.1, we modify Feige's lightest bin committee election protocol [14] to select such disjoint committees, where we carefully choose the parameters so that when eliminating recurring honest parties, we have the guarantee that (with overwhelming probability) we are left with enough honest parties.

2 Background and Definitions

2.1 Verifiable Secret Sharing

A *secret sharing* scheme, a notion introduced by Shamir [29], is a protocol that allows a dealer who holds a secret input s, to share his secret among n parties. The guarantee is that even if t of the parties are malicious, they gain no information about the secret s. A *verifiable secret sharing* (VSS) scheme, introduced by Chor et al. [10], is a secret sharing scheme with the additional guarantee that after the sharing phase, a dishonest dealer is either rejected, or is committed to a single secret s, that the honest parties can later reconstruct. Further, if the dealer is honest, then the original secret will be reconstructed, even if dishonest parties do not provide their correct shares.

Definition 2.1 (Verifiable Secret Sharing). *A* VSS protocol tolerating t malicious parties *for parties $\mathcal{P} = \{P_1, ..., P_n\}$ is a two-phase protocol* $(\mathsf{Share}, \mathsf{Rec})$*, where a distinguished dealer $P^* \in \mathcal{P}$ holds an initial input s, such that the following conditions hold for any adversary controlling at most t parties:*

– **Reconstruction:** *After the sharing phase, there exists a value s' such that all honest parties output s' in the reconstruction phase.*
– **Validity:** *If the dealer is honest, then $s' = s$.*
– **Secrecy:** *If the dealer is honest, then at the end of the sharing phase the joint view of the malicious parties is independent of the dealer's input s.*

2.2 Robust Multi-source Extractors

A multi-source extractor takes as input several independent sources, each with sufficient amount of entropy, and outputs a string that is statistically close to uniform. In this work, we need a multi-source extractor that extracts randomness even if some of the sources are "malicious," but independent of the "honest" ones. Such an extractor, which we refer to as a robust multi-source extractor, was constructed by Kamp, Rao, Vadhan and Zuckerman [23]. The notion of entropy that is used is *min-entropy.* A random variable $X \subseteq \{0,1\}^n$ is said to have min entropy k, denoted by $H_\infty(X) = k$, if for every $x \in \{0,1\}^n$, $\Pr[X = x] \leq \frac{1}{2^k}$, and is said to have min-entropy rate α if $H_\infty(X) \geq \alpha n$.

Theorem 2.2 ([23]). *For any constant $\delta > 0$ and every $n \in \mathbb{N}$, there is a polynomial-time computable robust multi-source extractor* $\mathsf{Ext} : \left(\{0,1\}^d\right)^n \to \{0,1\}^m$ *that takes as input n independent sources, each in $\{0,1\}^d$, and produces an m-bit string that is ϵ-close to uniform, as long as the min-entropy rate of the combined sources is δ, and where $m = 0.99\delta nd$ and $\epsilon = 2^{-\Omega((nd)/\log^3(nd))}$.*

2.3 Feige Committee Election Protocol

Our leakage-resilient coin tossing protocol uses Feige's lightest bin committee election protocol as a subroutine [14]. Feige's protocol gives a method for selecting a committee of approximately k parties (out of n) for a given parameter k.

It consists of one round, in which each party chooses and broadcasts a random bin in $\left[\frac{n}{k}\right]$. The committee consists of the parties in the lightest bin.

Lemma 2.3 ([14]). *For any constant $\beta > 0$ and any $k < n$, Feige's protocol is a 1-round public-coin protocol that elects a committee $\mathcal{E}$ such that for any set $\mathcal{C} \subset [n]$ of size $t = \beta n$, it holds that $|\mathcal{E}| \leq k$, and $\forall$ constant $\epsilon > 0$,* $\Pr[|\mathcal{E} \setminus \mathcal{C}| \leq (1-\beta-\epsilon)k] < \frac{n}{k}\exp[-\frac{\epsilon^2 k}{2(1-\beta)}]$ *and* $\Pr\left[|\mathcal{E} \cap \mathcal{C}|/|\mathcal{E}| \geq \beta + \epsilon\right] < \frac{n}{k}\exp[-\frac{\epsilon^2 k}{2(1-\beta)}]$.

3 Modeling Leakage in Distributed Protocols

We consider synchronous point-to-point networks with a broadcast channel. Point-to-point channels are assumed to be authenticated and to provide partial privacy guarantees (see discussion below). We consider n-party protocols where up to t statically corrupted parties perform arbitrary malicious faults. More precisely, we consider a computationally unbounded adversary who sees the internal state of all corrupted parties and controls their actions. We also assume the adversary is *rushing*, i.e. in any round he can wait until all honest parties send their messages before selecting the messages of corrupted parties. Our results hold information theoretically, with no computational assumptions.

In this work we propose a strengthening of the standard model, where in addition the adversary is able to *leak* a constant fraction of information on the secret state of each (honest) party. We model this by allowing the adversary to adaptively make leakage queries (i, f) throughout the protocol, where $i \in [n]$ and $f : \{0,1\}^* \to \{0,1\}$, and giving him the evaluation of f on the secret state of party i. Note that this also captures leakage on communication channels, as parties' messages are computed as a function of public information and their personal secret state; thus, we do not need to assume fully private channels, but rather channels that achieve privacy with bounded information leakage.

For simplicity, we consider length-bounded leakage. Namely, we require that no more than $\lambda|\mathsf{state}_i|$ leakage queries can be made on any single party i's secret state for some leakage rate λ, where $|\mathsf{state}_i|$ denotes the maximal size of the secret state of party i at any given time during the protocol. But, our constructions work equally well in the more general model of [27] where the output length of the leakage on state_i is not restricted, as long as the entropy of state_i is decreased by no more than the fraction λ.

Note that in this model, each leakage query is applied to the secret state of a single party. Since participants of a distributed protocol typically run on different physical hardware (and in fact in many cases in different locations across the world), it is reasonable to assume each physical attack reveals information about one party's execution. To maximize generality within this setting, we allow leakage queries on different parties' secret states to be interleaved (i.e., leak from party i, then from party j, and then again from party i), and to be adaptively selected as a function of prior leakage.

We refer to such an adversary who can corrupt t parties and leak λ fraction from the secret state of each honest party as a (t, λ) *adversary*, and say that a distributed protocol is (t, λ) *leakage resilient* if its original properties are satisfied against such an adversary. In this paper, we will focus on constructing a leakage-resilient unbiased coin tossing protocol.

Definition 3.1 (Leakage-Resilient Distributed Coin Tossing). *A protocol for parties $\mathcal{P} = \{P_1, ..., P_n\}$ is a (t, λ)* leakage-resilient m-bit distributed coin tossing protocol *if the following conditions hold for any (t, λ) adversary:*

- **Agreement:** *At the conclusion of the protocol, each party outputs a value $v_i \in \{0,1\}^m$. For all honest parties P_i, P_j, it holds that $v_i = v_j$.*
- **Randomness:** *With overwhelming probability in n (even if malicious parties abort prematurely), the distribution of the honest output value v given the view of the adversary is statistically close to uniform in $\{0,1\}^m$.*

4 Verifiable Secret Sharing with Leakage

One of the subroutines in our leakage-resilient coin tossing protocol is a protocol achieving verifiable secret sharing (VSS) in the presence of leakage. Recall the standard VSS guarantee is that for any adversary $\mathcal{A}$ who corrupts t parties, a dishonest dealer is committed to a single secret which will be reconstructed by honest parties, and the secret input s of an honest dealer remains secret given the view of $\mathcal{A}$ at the conclusion of the sharing phase. For our purposes, we will need stronger guarantees, where for any (t, λ) adversary $\mathcal{A}$ who corrupts t parties *and leaks* λ-fraction of each honest party's secret state (including the dealer's), the VSS reconstruction property still holds, and the secret input s of an honest dealer retains a constant fraction of its original entropy given the entire view of $\mathcal{A}$ (including leakage). We refer to this property as weak leakage resilience.

Definition 4.1 (WLR-VSS). *A* (λ, ϵ)-weakly leakage-resilient VSS protocol tolerating t malicious parties *for parties $\mathcal{P} = \{P_1, ..., P_n\}$ is a VSS protocol such that for any (t, λ) adversary $\mathcal{A}$, with overwhelming probability in n, the VSS reconstruction and validity properties hold, in addition to the following modified secrecy property: If the dealer is honest during the sharing phase, then with overwhelming probability over the view $y \leftarrow \mathsf{view}_{\mathcal{A}}(S)$ of $\mathcal{A}$ at the conclusion of the sharing phase of the protocol, $H_\infty(S|\mathsf{view}_{\mathcal{A}}(S) = y) \geq \epsilon H_\infty(S)$.*

To construct a WLR-VSS protocol, we use a modified version of the Shamir secret sharing scheme [29]. Recall in Shamir's scheme, a secret $s \in \mathbb{F}$ is shared by sampling a random degree d polynomial $p(x) = \sum_{j=0}^{d} a_j x^j \in \mathbb{F}[x]$ such that $a_0 = s$, and taking the shares s_i to be the evaluations $p(i)$. The scheme is secure against d malicious parties, and d is typically set to t. In our case, we embed a (large) secret $s \in \mathbb{F}^{d-t+1}$ as the first *several* coefficients of a polynomial $p(x)$ of degree $d = n - 2t - 1$.[3] The last t coefficients are chosen randomly, and the secret shares are evaluations $s_i = p(i)$. Intuitively, any t shares are independent of s, and with high probability each bit of leakage (on any party) will not decrease

[3] In our case, we will consider $n = (3 + \delta)t$, which yields $d = (1 + \delta)t - 1$, and $s \in \mathbb{F}^{\delta t}$.

the entropy of s by significantly more than this amount. The structure of secret shares as a Reed-Solomon code with minimum distance $n - d = 2t + 1$ further implies that the true secret can be reconstructed even if t of the parties are malicious [24]. We will denote this secret sharing scheme by $(\mathsf{SS}, \mathsf{RecSS})$.

Our WLR-VSS protocol is inspired by [4]. First, the dealer secret shares his input s via SS, along with two additional random values r, r'. Using the additive homomorphic property of SS, the parties check the dealer by broadcasting a (randomly selected) linear combination of their given shares, and verifying that together they form a valid codeword. To protect an honest dealer from being disqualified due to malicious parties giving bad values, the dealer will broadcast the true shares of complaining parties, and these values will be verified in a second check of the same form.

Loosely, since a dishonest dealer does not know what linear combination will be chosen, it is unlikely that he can distribute bad shares that pass these tests. Leakage information will not help, as the only secret values in the protocol are the distributed shares, which the dealer already knows (in fact, chooses) himself. On the other hand, no information on an *honest* dealer's secret s is revealed from the linear combinations, since shares of s are masked by shares of the random r, r'. So the only information learned about s comes from leakage, which leaves sufficient entropy remaining by the properties of SS.

Let $\mathbb{F}$ be a field with $\log |\mathbb{F}| = 2n$. We define $(\mathsf{Share}_{\mathsf{WLR}}, \mathsf{Rec}_{\mathsf{WLR}})$ in Figure 1.

Theorem 4.2. *Let $n = (3+\delta)t$ for some constant $\delta > 0$. Then for any constants $\epsilon < 1$ and $\lambda \leq \frac{\delta(1-\epsilon)}{10+6\delta}$, the protocol $(\mathsf{Share}_{\mathsf{WLR}}, \mathsf{Rec}_{\mathsf{WLR}})$ is a (λ, ϵ)-weakly leakage-resilient VSS protocol tolerating t malicious parties that runs in $O(1)$ rounds.*

Due to space limitations, we defer the proof to the full version [6].

WLR-VSS is sufficient for our coin-tossing construction; however, some applications may demand a stronger notion of leakage resilience, where given the view of a (t, λ) adversary, the secret s retains its *full* entropy. Note that this requirement is impossible to achieve for any standard VSS protocol, as the adversary can always leak information on s directly from the dealer. We thus put forth a new notion of *oblivious* secret sharing, where the dealer shares a uniformly distributed secret whose value *he does not know.*

Definition 4.3 (Leakage-Resilient Oblivious (LRO) VSS). *A* λ-leakage-resilient oblivious (LRO) VSS protocol tolerating t malicious parties *for parties $\mathcal{P} = \{P_1, ..., P_n\}$ is a VSS protocol with no validity requirement, such that for any (t, λ) adversary $\mathcal{A}$, with overwhelming probability in n, the VSS reconstruction property holds, in addition to the following modified secrecy property: If the dealer is honest during the sharing phase, then with overwhelming probability over the view $y \leftarrow \mathsf{view}_{\mathcal{A}}$ of $\mathcal{A}$ at the conclusion of this phase, $\mathsf{view}_{\mathcal{A}}$ is independent of the value s' that will be reconstructed in the second phase.*

In the full version [6], we present a λ-LRO-VSS protocol for $\lambda = \Omega(1)$, tolerating $t \leq \frac{n}{3+\delta}$ malicious parties (for any constant $\delta > 0$), using a WLR-VSS protocol as a black box. Loosely, the dealer samples and WLR-VSSes a random value

$\mathsf{Share}_{\mathsf{WLR}}(s)$:

Round 1: The dealer P^* selects two random values $r, r' \leftarrow \mathbb{F}^{\delta t}$, and runs three independent executions of the modified polynomial secret sharing algorithm (see above): $(s_1, ..., s_n) \leftarrow \mathsf{SS}(n, t, s)$, $(r_1, ..., r_n) \leftarrow \mathsf{SS}(n, t, r)$, $(r'_1, ..., r'_n) \leftarrow \mathsf{SS}(n, t, r')$. To each party i, P^* sends the corresponding three shares s_i, r_i, and r'_i.

Round 2: Each party P_i broadcasts three random pairs of bits $\alpha_i, \beta_i, \gamma_i \in \{0,1\}^2$. Take α, β, γ to be the corresponding elements in $\mathbb{F}$ with bit descriptions $(\alpha_1, ..., \alpha_n)$, $(\beta_1, ..., \beta_n)$, $(\gamma_1, ..., \gamma_n) \in \{0,1\}^{2n}$. (Recall $\log|\mathbb{F}| = 2n$).

Round 3: Each P_i broadcasts the linear combination of his shares $\alpha s_i + \beta r_i + \gamma r'_i \in \mathbb{F}$.

Round 4: Consider the received vector $v = (v_1, ..., v_n)$, where supposedly $v_i = \alpha s_i + \beta r_i + \gamma r'_i \; \forall i$.

- If v is a valid codeword (i.e., all points lie on a degree-d polynomial), the dealer is accepted, and the sharing phase concludes.
- If v is distance $> t$ away from a valid codeword, the dealer is rejected, and the sharing phase concludes.
- Otherwise, let $D \subset [n]$ be the components i in disagreement with the nearest codeword. For each $i \in D$, P^* broadcasts all three shares s_i, r_i, r'_i. If any linear combination $\alpha s_i + \beta r_i + \gamma r'_i$ with $i \in D$ is inconsistent with the nearest codeword, the dealer is rejected. Otherwise, parties continue to the next step.

Rounds 5-6: Repeat Rounds 2-3. That is, each party broadcasts a new triple of random $\tilde{\alpha}_i, \tilde{\beta}_i, \tilde{\gamma}_i$ and then broadcasts the linear combination $\tilde{v}_i = \tilde{\alpha} s_i + \tilde{\beta} r_i + \tilde{\gamma} r'_i$ of his shares.

Local Computation: Consider the new vector $\tilde{v}$ of values received in Round 6, where $\forall i \in D$ we use the values $(s_, r_i, r'_i)$ broadcast by the dealer in Round 4.

- If $\tilde{v}$ is distance $> t$ away from a valid codeword, the dealer is rejected.
- Otherwise, let $\tilde{D}$ be the set of parties whose components differ from the codeword closest to $\tilde{v}$. If $D \cap \tilde{D} \neq \emptyset$ or $|D \cup \tilde{D}| > t$, then the dealer is rejected. Otherwise, the dealer is accepted.

$\mathsf{Rec}_{\mathsf{WLR}}()$:

Round 1: Each party P_i broadcasts his share s_i.

Local Computation: Locally, P_i runs the modified polynomial secret sharing reconstruction algorithm $s' \leftarrow \mathsf{RecSS}(s'_1, ..., s'_n)$, where s'_i is the value broadcast by party P_i, and outputs this value s'.

Fig. 1. Weakly leakage-resilient VSS protocol, $(\mathsf{Share}_{\mathsf{WLR}}, \mathsf{Rec}_{\mathsf{WLR}})$

x, erases it, then samples and WLR-VSSes a second random value y; the final output will be $\mathsf{Ext}_2(x, y)$, where Ext_2 is a two-source extractor. (Note we do not require perfect erasures, by treating remaining information as leakage). To ensure information is never leaked on x and y together, their values will be shared among two disjoint committees, which are selected by all parties.

5 Disjoint Committee Election

We now exhibit a 1-round public-coin protocol for electing $m = \log^2 n$ *disjoint* "good" committees $\mathcal{E}_1, ..., \mathcal{E}_m$ of size approximately $n^{1/2}$.

Let $m = \log^2 n$ and $k = n^{1/2}$. We run m parallel repetitions of the Feige lightest bin protocol with $\frac{n}{k}$ bins (see Section 2.3), where recurring parties are removed from all but the first committee in which they appear. More explicitly, define the protocol ElectDisj as follows. In a single round, each party P_i broadcasts m random values $r_1^i, ..., r_m^i \leftarrow \left[\frac{n}{k}\right]$. Locally, everyone iterates through $j = 1, ..., m$, setting $\mathcal{E}_j$ to be the parties in the lightest bin in the jth election, defined by $r_j^1, ..., r_j^n$. Then, to force disjointness, all parties in $\mathcal{E}_j \cap \left(\bigcup_{j'<j} \mathcal{E}_{j'}\right)$ are removed from $\mathcal{E}_j$, and this becomes the final jth elected committee.

Proposition 5.1. *The protocol* ElectDisj *is a 1-round public-coin protocol for electing $m = \log^2 n$ committees $\mathcal{E}_i$ such that for any constants $\beta, \epsilon > 0$, and any set of corrupted parties $\mathcal{C} \subset [n]$ of size βn, the following events simultaneously occur with overwhelming probability in n:* (1) $\forall i \neq j,\ \mathcal{E}_i \cap \mathcal{E}_j = \emptyset$, (2) $\forall i,\ (1 - \beta - \epsilon)n^{1/2} \leq |\mathcal{E}_i| \leq n^{1/2}$, (3) $\forall i,\ \frac{|\mathcal{E}_i \cap \mathcal{C}|}{|\mathcal{E}_i|} < \beta + \epsilon$.

By construction, property (1) holds immediately. Further, by Lemma 2.3, each $\mathcal{E}_j$ is of size at most k and has at least $(1-\beta-\frac{\epsilon}{2})k$ honest parties before erasures. It thus remains to analyze the number of players who will appear in multiple committees, and to show that erasing does not decrease the number of honest parties in any committee by too much. We refer the reader to the full version of the paper for a complete proof [6].

6 Unbiased Coin Tossing with Leakage

In this section, we construct our final leakage-resilient coin tossing protocol, as characterized by Definition 3.1. Our construction makes black-box use of the tools developed in the previous sections: in particular, a weakly leakage-resilient verifiable secret sharing (WLR-VSS) protocol (from Section 4), and a disjoint committee election protocol (from Section 5).

Recall we are within the model of a synchronous point-to-point network with broadcast, and channels are assumed to be authenticated and private (with leakage). Our results are information theoretic, without cryptographic assumptions.

Theorem 6.1. *For any constants $\delta, \epsilon > 0$, any $\lambda \leq \frac{\delta(1-\epsilon)}{10+6\delta}$, any $n \geq (3+\delta)t$, and any m, there exists a λ-leakage-resilient n-party distributed coin tossing protocol tolerating t malicious parties that generates m unbiased random bits, and terminates in $O(1)$ rounds.*

Proof. Let δ' be any constant such that $\delta' < \delta$. In Figure 2, we construct the desired coin tossing protocol CoinToss using the following tools:

Elect: Feige's 1-round public-coin protocol to elect a primary committee of size approximately $\log^2 n$, as in Lemma 2.3.

ElectDisj: a 1-round public-coin protocol for electing $\log^2 n$ *disjoint* secondary committees of size $n' \approx n^{1/2}$,[4] as in Proposition 5.1.

[4] Note that we will use prime notation (e.g., n', t', δ') to denote parameters pertaining to the secondary committees.

CoinToss:

Step 1: Run Elect to elect a primary committee of approximate size $\log^2 n$ (see Lemma 2.3). Denote the set of indices of elected parties by $\mathcal{E} \subset [n]$.

Step 2: Run the ElectDisj protocol on the remaining parties $[n] \setminus \mathcal{E}$ to elect $|\mathcal{E}|$ disjoint secondary committees $\mathcal{E}'_1, ..., \mathcal{E}'_{|\mathcal{E}|}$, each of size approximately $n^{1/2}$ (see Prop. 5.1).

Step 3: $\forall i \in \mathcal{E}$, P_i samples a random value $r_i \leftarrow \mathbb{F}^{\delta' t'}$ and verifiably secret shares it among the parties in his corresponding secondary committee, $\mathcal{E}'_i$. That is, he acts as a dealer in an execution of $\mathsf{Share}_{\mathsf{WLR}}(r_i)$.

Step 4: For each $i \in \mathcal{E}$, all parties in the secondary committee $\mathcal{E}'_i$ execute the reconstruction phase $r_i \leftarrow \mathsf{Rec}_{\mathsf{WLR}}()$ on the shares dealt by P_i. For any party $i \in \mathcal{E}$ who was rejected as a dealer in the previous step, set $r_i = 0$. Each secondary committee member broadcasts his reconstructed value for r_i.

Local Computation: Let r_i^* be the most common value received from the parties in secondary committee $\mathcal{E}'_i$ in the previous step. Output $r \leftarrow \mathsf{Ext}(\{r_i^*\}_{i\in\mathcal{E}})$.

Fig. 2. Leakage-resilient coin tossing protocol

($\mathsf{Share}_{\mathsf{WLR}}, \mathsf{Rec}_{\mathsf{WLR}}$): a (λ, ϵ) WLR-VSS protocol for n' parties, tolerating $t' \leq \frac{n'}{3+\delta'}$ malicious parties, terminating in $O(1)$ rounds, as in Theorem 4.2.

$\mathsf{Ext} : (\{0,1\}^d)^{\log^2 n} \to \{0,1\}^m$: a robust multi-source extractor, where $m = .99(\frac{2}{3}\log^2 n)(\epsilon d)$, as in Theorem 2.2. We interpret each element $\{0,1\}^d$ as an element of $\mathbb{F}^{\delta' t'}$ (i.e., $d = \delta' t' \log|\mathbb{F}|$), where the size of $\mathbb{F}$ depends on the desired output length m.

By Proposition 5.1, with overwhelming probability in n, the disjoint secondary committees $\mathcal{E}'_i$ will be "good," in that they each have size $n^{1/2-\zeta} \leq |\mathcal{E}'_i| \leq n^{1/2}$ for any constant $\zeta > 0$ and it holds that $n'_i \geq (3+\delta')t'_i$, where $n'_i = |\mathcal{E}'_i|$ and $t'_i = |\mathcal{E}'_i \cap \mathcal{C}|$ (where $\mathcal{C}$ is the set of corrupted parties). We will thus assume this is the case. Since $n'_i \geq (3+\delta')t'_i$, the validity, reconstruction, and secrecy properties of the (λ, ϵ) WLR-VSS protocol (see Definition 4.1) will hold for the ith execution of ($\mathsf{Share}_{\mathsf{WLR}}, \mathsf{Rec}_{\mathsf{WLR}}$) with overwhelming probability in n'_i (and thus in n). We now show that the protocol CoinToss satisfies the desired agreement and randomness properties (see Definition 3.1).

Agreement. By the reconstruction property of the WLR-VSS protocol, for each $P_i \in \mathcal{E}$, the honest parties in $\mathcal{E}'_i$ will agree on the reconstructed value $r_i \leftarrow \mathsf{Rec}_{\mathsf{WLR}}()$ and will broadcast this value to all parties in Step 4 (where $r_i = 0$ if P_i was rejected as a dealer in the sharing phase of the VSS). Since a majority of the parties in $\mathcal{E}'_i$ are honest, all honest parties in $[n]$ will agree on $r_i^* = r_i$ for each i, and so will agree on the final output r.

Randomness. Consider the values r_i reconstructed by each secondary committee $\mathcal{E}'_i$. By the reconstruction property of the WLR-VSS, each r_i is fully determined by the conclusion of the sharing phase (Step 3 of the CoinToss protocol). The secrecy property of the WLR-VSS implies that at the end of the sharing phase, even given the view of the adversary ($\mathsf{view}_{\mathcal{A}}$), each *honest* party's r_i retains at least $\epsilon \cdot (\delta' t' \log |\mathbb{F}|)$ bits of entropy. Therefore, conditioned on $\mathsf{view}_{\mathcal{A}}$ (which includes leakage), the random variables $r_1^*, ..., r_{|\mathcal{E}|}^* \in \mathbb{F}^{\delta' t'}$ are independent, where for all $j \in \mathcal{E} \cap \mathcal{C}$ we think of r_j^* as fixed. Further, together they have total min-entropy at least $(|\mathcal{E} \setminus \mathcal{C}|)(\epsilon\delta' t' \log |\mathbb{F}|)$. By Lemma 2.3, $|\mathcal{E} \setminus \mathcal{C}| \geq (1 - \frac{1}{3+\delta} - \zeta) \log^2 n$ for any constant $\zeta > 0$ with overwhelming probability in n. Since the extractor we use can extract even when many of the sources r_j^* are fixed, we can simply take the loose bound $|\mathcal{E} \setminus \mathcal{C}| \geq \frac{2}{3} \log^2 n$. By Theorem 2.2, the final output $r = \mathsf{Ext}(\{r_i^*\}_{i \in \mathcal{C}})$ will be statistically close to uniform over $\{0,1\}^m$ with $m = .99(\frac{2}{3} \log^2 n)(\epsilon\delta' t' \log |\mathbb{F}|)$.

References

1. Akavia, A., Goldwasser, S., Hazay, C.: Distributed public key schemes secure against continual leakage (2010) (manuscript)
2. Akavia, A., Goldwasser, S., Vaikuntanathan, V.: Simultaneous hardcore bits and cryptography against memory attacks. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 474–495. Springer, Heidelberg (2009)
3. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: PODC, pp. 27–30 (1983)
4. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, pp. 1–10 (1988)
5. Bitansky, N., Canetti, R., Goldwasser, S., Halevi, S., Kalai, Y., Rothblum, G.: Program obfuscation with leaky hardware (manuscript, 2011)
6. Boyle, E., Goldwasser, S., Kalai, Y.T.: Leakage-resilient coin tossing. Cryptology ePrint Archive, Report 2011/291 (2011), http://eprint.iacr.org/
7. Bracha, G.: An asynchronous $[(n-1)/3]$-resilient consensus protocol. In: PODC, pp. 154–162 (1984)
8. Canetti, R., Feige, U., Goldreich, O., Naor, M.: Adaptively secure multi-party computation. In: STOC, pp. 639–648 (1996)
9. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: STOC, pp. 11–19 (1988)
10. Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults. In: FOCS, pp. 383–395 (1985)
11. Cleve, R.: Limits on the security of coin flips when half the processors are faulty (extended abstract). In: STOC, pp. 364–369 (1986)
12. Dwork, C., Shmoys, D.B., Stockmeyer, L.J.: Flipping persuasively in constant time. SIAM J. Comput. 19(3), 472–499 (1990)
13. Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: FOCS (2008)
14. Feige, U.: Noncryptographic selection protocols. In: FOCS (1999)
15. Feldman, P., Micali, S.: Byzantine agreement in constant expected time (and trusting no one). In: FOCS, pp. 267–276 (1985)
16. Garg, S., Jain, A., Sahai, A.: Leakage-resilient zero knowledge. In: Advances in Cryptology – CRYPTO 2011 (To appear, 2011)
17. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC (1987)
18. Goldwasser, S., Micali, S.: Probabilistic encryption and how to play mental poker keeping secret all partial information. In: STOC, pp. 365–377 (1982)

19. Goldwasser, S., Sudan, M., Vaikuntanathan, V.: Distributed computing with imperfect randomness. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 288–302. Springer, Heidelberg (2005)
20. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003)
21. Kalai, Y.T., Li, X., Rao, A.: 2-source extractors under computational assumptions and cryptography with defective randomness. In: FOCS, pp. 617–626 (2009)
22. Kalai, Y.T., Li, X., Rao, A., Zuckerman, D.: Network extractor protocols. In: FOCS, pp. 654–663 (2008)
23. Kamp, J., Rao, A., Vadhan, S., Zuckerman, D.: Deterministic extractors for small-space sources. In: STOC, pp. 691–700 (2006)
24. McEliece, R.J., Sarwate, D.V.: On sharing secrets and reed-solomon codes. Commun. ACM 24, 583–584 (1981)
25. Micali, S., Reyzin, L.: Physically observable cryptography. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 278–296. Springer, Heidelberg (2004)
26. Moran, T., Naor, M., Segev, G.: An optimally fair coin toss. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 1–18. Springer, Heidelberg (2009)
27. Naor, M., Segev, G.: Public-key cryptosystems resilient to key leakage. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 18–35. Springer, Heidelberg (2009)
28. Rabin, M.O.: Randomized byzantine generals. In: FOCS, pp. 403–409 (1983)
29. Shamir, A.: How to share a secret. Commun. ACM 22, 612–613 (1979)

Brief Announcement: Composition Games for Distributed Systems: The EU Grants Games[⋆]

Shay Kutten, Ron Lavi, and Amitabh Trehan

Faculty of IE&M, The Technion, Haifa, 32000 Israel
{Kutten,ronlavi}@ie.technion.ac.il, amitabh.trehaan@gmail.com

A traditional distributed system was, usually, designed by some centralized manufacturer and owned by some central owner. On the other hand, many modern distributed systems (e.g., many Peer to Peer (P2P) networks) are formed when people team up to pool their resources together to form such a system. We aim to initiate an investigation into the way people make a distributed decision on the composition of such a system, with the goal of realizing high values. Intuitively, we look at settings in which, by teaming up, a node increases its utility, however, it also pays a cost that often (as mentioned later) increases with the size of the system. The right balance is achieved by the right size system.

We terms such settings "European Union Grant Games", motivated by the following case study. In FP7, the current, 7th Framework Programme for supporting research in Europe, the main emphasis is put on forming sets of researchers, mostly large sets. A rationale is that a large impact can be realized by a large set of researchers- a "network of excellence". Of course, what the commission is really interested in is the combination of size and quality. Note that researchers also have an opposing motivation to form small sets, since the grant is divided among set members. These factors also exist when people come together to form certain distributed systems. For example, in the formation of some social groups, people come together of their own free will, and realize benefit by pooling their resources together; yet at the same time they want to limit the size of their groups to maintain 'quality' and to reduce overhead and risks.

As opposed to the studies of network creation games [2], we deal with *splitting* a community, rather than with creating connections. As opposed to the classic game theoretic work on group formation e.g. [3], we deal with the quality of the groups formed. We study price of anarchy(POA) [4,5] (and also strong price of anarchy(SPOA) [1]) – the ratio between the average (over the system's components) value of the optimal possible system, and the average value for the system formed in the worst equilibrium. We formulate and analyze games showing how simple changes in protocol can affect the SPOA drastically. We identify two important properties for a low SPOA: whether the parties joining a group need to

[⋆] Longer version at *http://arxiv.org/abs/1105.5255*. This paper supported in part by the Net-HD MAGNET consortium, by the Israel Science Foundation, by the Technion Gordon Center, the Bi-national US-Israel Science Foundation, the Israeli ministry of science, the Google Inter-university center for Electronic Markets and Auctions, and at the Technion by a fellowship of the Israel Council for Higher Education.

D. Peleg (Ed.): DISC 2011, LNCS 6950, pp. 197–199, 2011.

reach agreement, and whether they are given an option of appealing a rejection from a system. We show that the latter property is especially important if there are some pre-existing collaboration/communication constraints.

The general setting: A granting agency wishes to award a prize of M dollars to a subset formed from n researchers. Each researcher i has a value v_i representing her overall quality. A subset whose sum of values is at least some given threshold T is an *eligible subset*. We assume $v_i < T$ for every researcher i,at least one eligible subset, and the agency does not know the researcher values *a priori* but can verify the values from the grant proposal. We construct protocols for the agency, by which researchers form candidate *consortia* (i.e. subsets); the agency wishes to maximize the average value of the winning consortium. The prize is equally distributed among the winning researchers.

Results. We study three games in order of improving quality. A first "naive" attempt is the GOLD-RUSH GAME: Informally (details in full paper), the agency asks each researcher to choose a label from a finite set and submit it with a proposal. Researchers with the same label form a consortium and the prize goes to the consortium with the highest average. Thus, the player strategy is to choose a label and her utility is the shared winning if she wins. The Gold-Rush game has a bad POA ($n/2$): maybe it was "too easy" for anybody to join a consortium.

THE MAIN GAMES: The next game has stricter rules for joining a group. Informally, all consortium members must agree on the consortium membership. We further consider the case where researchers are nodes of a graph and are connected by an edge if they cooperate directly; each consortium is required to be a connected component in this graph. The SPOA is improved but is not optimal. Intuitively, this game (CCC-CN) suffers from being TOO strict. We then introduce a new game with looser rules (but stricter than Gold-rush). Informally, the first round of this game is like the CCC-CN game, but in subsequent rounds, subsets of researchers can appeal to join the winners if they can prove that they improve the average. Eventually, a fixed point is achieved. This last game shows strong improvement over the other games (results are in the full paper). However, SPOA can vary over different topologies and there is an interesting relationship between some graph parameters and SPOA; For example, it is intriguing that SPOA grows in opposite directions for the complete graph and line graph .

References

1. Andelman, N., Feldman, M., Mansour, Y.: Strong price of anarchy. Games and Economic Behavior 65(2), 289–317 (2009); Preliminary version in SODA 2007
2. Fabrikant, A., Luthra, A., Maneva, E., Papadimitriou, C.H., Shenker, S.: On a network creation game. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, PODC (2003)

3. Konishi, H., Le Breton, M., Weber, S.: Pure strategy Nash equilibrium in a group formation game with positive externalities. Games and Economic Behavior 21, 161–182 (1997)
4. Koutsoupias, E., Papadimitriou, C.: Worst-case equilibria. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, pp. 404–413. Springer, Heidelberg (1999)
5. Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.: Algorithmic game theory. Cambridge University Press, Cambridge (2007)

Brief Announcement: Distributed Approximations for the Semi-matching Problem*

Andrzej Czygrinow[1], Michal Hanćkowiak[2], Krzysztof Krzywdziński[2], Edyta Szymańska[2], and Wojciech Wawrzyniak[2]

[1] School of Mathematical and Statistical Sciences, Arizona State University, Tempe, AZ,85287-1804, USA
andrzej@math.la.asu.edu

[2] Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Poznań, Poland
{mhanckow,kkrzywd,edka,wwawrzy}@amu.edu.pl

We consider the *semi-matching* problem in bipartite graphs. The network is represented by a bipartite graph $G = (U \cup V, E)$, where U corresponds to clients, V to servers, and E is the set of available connections between them. The goal is to find a set of edges $M \subseteq E$ such that every vertex in U is incident to exactly one edge in M. The *load* of a server $v \in V$ is defined as the square of its degree in M and the problem is to find an optimal semi-matching, i.e. a semi-matching that minimizes the sum of the loads of the servers. Formally, given a bipartite graph $G = (U \cup V, E)$, a semi-matching in G is a subgraph M such that $\deg_M(u) = 1$ for every $u \in U$. A semi-matching M is called optimal if $\mathrm{cost}(M) := \sum_{v \in V} (\deg_M(v))^2$ is minimal. It is not difficult to see that for any semi-matching M, $\frac{|U|^2}{|V|} \leq \mathrm{cost}(M) \leq \Delta|U|$ where Δ is such that $\max_{v \in V} d(v) \leq \Delta$. Consequently, if M^* is optimal and M is arbitrary, then $\mathrm{cost}(M) \leq \frac{\Delta|V|\mathrm{cost}(M^*)}{|U|}$. Our main result shows that in some networks the $\frac{\Delta|V|}{|U|}$ factor can be reduced to a constant (Theorem [1]).

The semi-matching problem has been extensively studied under various names in the scheduling literature. Recently it has received renewed attention after the paper by Harvey, Ladner, Lovász, and Tamir [4], where the name *semi-matching* was introduced. In [4] and [2] the authors give two sequential polynomial time algorithms that find an optimal semi-matching in a graph. We have considered the distributed complexity of the semi-matching problem in the synchronous, message-passing model of computations (*Local*). In this model the nodes of the underlying network communicate in synchronized rounds. In a single round every vertex can send and receive messages from all of its neighbors and can perform some local computations. The running time of the algorithm is the number of rounds needed to solve a problem. Let M^* be an optimal semi-matching in $G = (U \cup V, E)$ and let $\Delta = \max_{v \in V} \deg(v)$.

* The research supported by grant N N206 565740.

Theorem 1. *There is a distributed algorithm that finds a semi-matching M in $G = (U \cup V, E)$ such that cost$(M) \leq \lambda \cdot$ cost(M^*), where $\lambda = \min\left(3 \cdot 10^5, \frac{\Delta|V|}{|U|}\right)$ and the time complexity of the algorithm is $O(\min\left(\Delta^2, \Delta \log^4(|V| + |U|)\right)$.*

One of the main appeals of Theorem 1 is the simplicity of the procedure that finds the semi-matching:

```
Algorithm
 1. Let $M := \emptyset$.
 2. For $i = 1$ to $\Delta$ do:
    (a) Find a maximal matching $M_i$ in $G_i$ using algorithms from [3].
    (b) Let $M := M \cup M_i$ and $G_{i+1} := G[V(G_i) \setminus (V(M_i) \cap U)]$.
 3. Return $M$.
```

It is not difficult to see that the outcome is a semi-matching in G. On the other hand, proving that M is a constant approximation of an optimal requires careful analysis. Although the approximation constant in Theorem 1 is most likely an artifact of our proof technique and can be significantly reduced, it is not possible to find an optimal solution efficiently in the *Local* model of computations. We have proved the following fact.

Theorem 2. *There exists a graph $G = (V, E)$ such that any deterministic distributed algorithm that finds an optimal semi-matching in G runs in $\Omega(|V|)$ rounds.*

The running time of the algorithm from Theorem 1 involves Δ and it is not clear if this dependency can be avoided. In that direction, we have proved that in some cases even when Δ is unbounded it is possible to find a good approximation in a constant number of rounds.

Theorem 3. *Let $c_1, c_2 \geq 2$ be such that $c_2 | c_1$ and c_2 is constant. Let $G = (U \cup V, E)$ be a bipartite graph such that for every $v \in V$, $d(v) = c_1$ and for every $u \in U$, $d(u) = c_2$. There is a distributed algorithm that finds a $(\min(3, c_2))$-approximation of an optimal semi-matching, in a constant number of rounds depending on c_2.*

The proof of Theorem 3 hinges on the fact that it is possible to find a semi-matching with cost of at most $3\frac{|U|^2}{|V|} \leq 3\text{cost}(M^*)$ where M^* is optimal.

The details of our work are available in [1].

References

1. Czygrinow, A., Hanćkowiak, M., Krzywdziński, K., Szymańska, E., Wawrzyniak, W.: Distributed approximation algorithm for the semi-matching problem (manuscript)
2. Fakcharoenphol, J., Laekhanukit, B., Nanongkai, D.: Faster algorithms for semi-matching problems (Extended abstract). In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 176–187. Springer, Heidelberg (2010)
3. Hanćkowiak, M., Karoński, M., Panconesi, A.: On the distributed complexity of computing maximal matchings. In: Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998), San Francisco, CA, USA, pp. 219–225 (January 1998)
4. Harvey, N.J.A., Ladner, R.E., Lovasz, L., Tamir, T.: Semi-matchings for bipartite graphs and load balancing. J. Algoritmss 59(1), 53–78 (2006)

Brief Announcement: Opportunistic Information Dissemination in Mobile Ad-Hoc Networks[⋆]: Adaptiveness vs. Obliviousness and Randomization vs. Determinism

Martín Farach-Colton[1], Antonio Fernández Anta[2], Alessia Milani[3], Miguel A. Mosteiro[1,4], and Shmuel Zaks[5]

[1] Department of Computer Science, Rutgers University, Piscataway, NJ, USA
{farach,mosteiro}@cs.rutgers.edu
[2] Institute IMDEA Networks, Madrid, Spain
antonio.fernandez@imdea.org
[3] LABRI, University of Bordeaux 1, Talence, France
milani@labri.fr
[4] LADyR, GSyC, Universidad Rey Juan Carlos, Móstoles, Madrid, Spain
[5] Department of Computer Science, Technion - Israel Institute of Technology, Haifa, Israel
zaks@cs.technion.ac.il

Abstract. In the context of Mobile Ad-hoc Networks (MANET), we study the problem of disseminating a piece of information, initially held by a source node, to some subset of nodes. We use a model of MANETs that is well suited for dynamic networks and opportunistic communication. We assume that network nodes are placed in a plane where they can move with bounded speed; they may start, crash and recover at different times; and they communicate in a collision-prone single channel. In this setup informed and uninformed nodes may be disconnected for some time, but eventually some informed-uninformed pair must be connected long enough to communicate. We show negative and positive results for different types of randomized protocols, and we contrast them with our previous deterministic results.

A MANET is a network of processing nodes that move in an environment that lacks any form of communication infrastructure. In this paper we revisit a class of MANETs that is well suited for opportunistic communication. Specifically, nodes are placed in a plane, in which they can move with bounded speed, and communication between nodes occurs over a collision-prone single channel. Our model includes a parameter α that mainly characterizes the connectivity and a parameter β that models the stability properties of the network, provided that

⋆ We thank Seth Gilbert for triggering the development of this work asking a question at DISC 2010. This research was partially supported by Spanish MICINN grant TIN2008-06735-C02-01, Comunidad de Madrid grant S2009TIC-1692, the National Science Foundation CCF-0937829 and by the ANR-project "ALADDIN".

nodes move, may crash and recover and may be activated at different times. These parameters characterize *any* model of dynamic network, and affect the progress that a protocol *may* achieve in solving basic tasks. Our model is formalized in [2] and it is slightly weaker than the one presented in [1]. In this context, we consider the problem of information dissemination. 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. To solve the Disseminationproblem we consider three classes of randomized algorithms: *locally adaptive* randomized algorithms where the probability of transmission of a node in a step of an execution may depend on its own communication history; *oblivious* randomized protocols where the probability of transmission of a node in a step of an execution depends only on predefined parameters; and *fair* randomized protocols, in which at each step all nodes transmit with the same probability.

Our Results. We have determined the minimum values for parameters α and β under which randomized protocols to disseminate information with large enough probability exist; we have studied the time complexity in relation with the maximum speed of movement and the probability of failure, and we have put the results obtained here in perspective of our results in [1] highlighting the impact of fundamental characteristics of Disseminationprotocols, such as determinism vs. randomization, and obliviousness vs. adaptiveness, on dissemination time. These results are extensively presented in [2] and summarized in Table 1. These results show that there is no gap between oblivious and locally adaptive protocols and that randomization reduces the time complexity of the problem in a linear factor in the oblivious case and in a logarithmic factor in the adaptive case (for reasonably small values of α).

Table 1. Oblivious and fair lower bounds to achieve success probability $p \geq 2^{-n/2}$. Locally adaptive lower bound in expectation. Upper bounds with probability $p \geq 1 - e^{-(n-1)/4}$.

		randomized	deterministic [1]
lower bounds	oblivious	$\Omega\left(\alpha n + n^2/\log n\right)$	$\Omega\left(\alpha n + n^3/\log n\right)$
	locally adaptive	$\Omega\left(\alpha n + n^2/\log n\right)$	$\Omega(\alpha n + n^2)$
	fair	$\Omega\left(\alpha n + n^2/\log n\right)$	–
upper bounds	oblivious	$O\left(\alpha n + (1+\alpha/\beta)\, n^2/\log n\right)$	$O(\alpha n + n^3 \log n)$
	locally adaptive	–	$O(\alpha n + n^2)$
	fair	$O\left(\alpha n + (1+\alpha/\beta)\, n^2/\log n\right)$	–

References

1. Fernández Anta, A., Milani, A., Mosteiro, M.A., Zaks, S.: Opportunistic Information Dissemination in Mobile Ad-hoc Networks: The Prot of Global Synchrony. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 374–388. Springer, Heidelberg (2010)
2. Farach-Colton, M., Fernández Anta, A., Milani, A., Mosteiro, M.A., Zaks, S.: Opportunistic Information Dissemination in Mobile Ad-hoc Networks: adaptiveness vs. obliviousness and randomization vs. determinism. Tech. rep. arXiv:1105.6151v1 (2011), `http://arxiv.org/abs/1105.6151`

Brief Announcement: Bridging the Theory-Practice Gap in Multi-commodity Flow Routing

Siddhartha Sen, Sunghwan Ihm,
Kay Ousterhout, and Michael J. Freedman

Princeton University

1 Introduction

In the *concurrent multi-commodity flow problem*, we are given a capacitated network $G = (V, E)$ of switches V connected by links E, and a set of commodities $\mathcal{K} = \{(s_i, t_i, d_i)\}$. The objective is to maximize the minimum fraction λ of any demand d_i that is routed from source s_i to target t_i. This problem has been studied extensively by the theoretical computer science community in the sequential model (e.g., [4]) and in distributed models (e.g., [2,3]). Solutions in the networking systems community also fall into these models (e.g., [1,6,5]), yet none of them use the state-of-the-art algorithms above. Why the gap between theory and practice? This work seeks to answer and resolve this question. We argue that existing theoretical models are ill-suited for real networks (§2) and propose a new distributed model that better captures their requirements (§3). We have developed optimal algorithms in this model for data center networks (§4); making these algorithms practical requires a novel use of programmable hardware switches. A solution for general networks poses an intriguing open problem.

2 Existing Models: Theory vs. Practice

Prior solutions to the multi-commodity flow problem fall in one of three models. In the *sequential model* [4], the entire problem input (solution) is known (computed) by a single entity. In the *Billboard model* [3], routing decisions are made by agents at the sources, one per commodity, that can read and write to a global "billboard" of link utilizations. In the *Routers model* [2], routing decisions are made locally by all switches by communicating only with their neighbors.

The fundamental problem with these models is that they are designed for a static demand matrix (i.e., a single set of commodities), whereas real systems must respond to rapidly changing demand matrices. The cost of collecting demands and communicating the flows in the sequential model makes it impractical to respond to changing demands at small timescales. Thus, systems like Hedera [1] recompute the flows from scratch at large scheduling intervals (seconds). Similarly, the cost of flooding link utilizations to the sources in the Billboard model causes systems like MPTCP [6] to apply only coarse congestion control at sources based on indirect information. The Routers model evades these problems, but because switches have no *a priori* knowledge of the network topology, flows may change direction or circulate repeatedly in the network. Thus, systems like FLARE [5] use pre-established routes and avoid rerouting altogether.

The second problem is that all known polynomial-time solutions, in all models, require fractionally splitting flows. Splitting flows causes packets to get reordered, which causes throughput to collapse in the TCP protocol. If a flow's paths have inconsistent latencies, queuing occurs at the target; such uncertain packet delivery times make it difficult to time retransmissions without exacerbating congestion. Thus, systems use either heuristics to solve the integer (unsplittable) multi-commodity flow problem [1], or complicated splitting heuristics that still cause reordering across subflows [6].

The third problem is that all models incorrectly assume that hardware switches are identical to end hosts. To forward traffic at line rate—1 or 10 Gbps for today's commodity switches—switches require high-speed matching on packet headers and offer limited general-purpose processing. Practical solutions must operate within these limits.

3 Routers Plus Pre-processing (RPP) Model

In almost any wired network, the demand matrix changes far more frequently than the network topology. Thus, we propose the following extension to the Routers model: *We allow arbitrary (polynomial-time) pre-processing of the network G at zero cost in time* (but charge for any space required to store the results). This decouples the problem of topology discovery from routing, turning the former into a *dynamic graph problem*.

We also introduce two novel issues of practicality. First, we allow $\mathrm{O}(1)$-sized messages that are injectively mapped to flow packets, or *in-band messages*, to be sent for free, and charge only for *out-of-band messages*. Second, when possible, we ensure paths of the same commodity are roughly equal in length, to minimize queuing and reordering at the target. We are interested in algorithms that are *partially asynchronous*, since otherwise we would need expensive synchronizers to simulate rounds.

4 Algorithms

We have devised a simple algorithm in the RPP model for data center fat-tree networks [1]. Our algorithm locally splits and rate-limits the aggregate demand to each target with the help of in-band messages, routing the maximum concurrent flow in an optimal $\mathrm{O}(H)$ parallel rounds, where H is the length of the longest flow path. By allowing approximate splitting, we can drastically reduce the amount of splitting in practice. Our solution uses carefully crafted rules in switches' forwarding tables that allow line-rate processing while minimizing packet reordering at the targets.

A solution for general networks in the RPP model poses an intriguing open problem. One approach may be to use the free pre-processing to initialize connectivity oracles that can route around "failed" (congested) links.

References

1. Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A.: Hedera: dynamic flow scheduling for data center networks. In: Proc. NSDI, pp. 281–296 (2010)
2. Awerbuch, B., Khandekar, R.: Distributed network monitoring and multicommodity flows: a primal-dual approach. In: Proc. PODC, pp. 284–291 (2007)

3. Awerbuch, B., Khandekar, R.: Greedy distributed optimization of multi-commodity flows. Distrib. Comput. 21(5), 317–329 (2009)
4. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. SIAM J. Comput. 37(2), 630–652 (2007)
5. Kandula, S., Katabi, D., Sinha, S., Berger, A.: Dynamic load balancing without packet reordering. SIGCOMM Comput. Commun. Rev. 37 (2007)
6. Wischik, D., Raiciu, C., Greenhalgh, A., Handley, M.: Design, implementation and evaluation of congestion control for multipath TCP. In: Proc. NSDI, pp. 99–112 (2011)

DISC 2011 Invited Lecture by Dahlia Malkhi: Going beyond Paxos

Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, and Ted Wobber

Microsoft Research, Silicon Valley

Talk Abstract

Flash storage is making inroads into data centers, enabling a new class of applications that require persistence as well as extreme performance. Clusters of flash can satisfy millions of I/Os per second at sub-millisecond latencies while consuming significantly less power than disk clusters. Unfortunately, current designs for scalable storage systems are predicated on the properties of hard disks, and can be inefficient or unreliable when used with flash clusters. New abstractions are required to fully realize the potential of flash clusters. We drew foundational lessons from designing Falcon, a new transactional storage cluster which works at the speed of flash. In this keynote, we describe the challenges and design choices we made.

The key idea in Falcon is to expose a cluster of network-attached flash devices as a single, shared log to clients running within the data center. Applications running on the clients can append data to this log or read entries from its middle. Internally, this shared log is implemented as a distributed log spread over the flash cluster. This design makes sense for two reasons:

– **Falcon is a distributed SSD...**
  Falcon runs over raw flash chips directly attached to the network, obviating the need for servers or SSD controllers in the storage cluster; this slashes infrastructure cost and power consumption by an order of magnitude. In effect, Falcon acts as a distributed SSD, implementing flash management and wear-leveling at cluster scale.
– **... with a shared log interface.**
  From a top-down perspective, the Falcon shared log is a powerful primitive for building applications that require strong consistency, such as databases, transactional key-value stores and metadata services.

Abstractly the storage cluster implements an append-only totally ordered log, and we need to implement a reliable and high-throughput total-ordering mechanism that stores entries in the log. The solution seemed obvious at first: Use State-Machine-Replication (SMR) and Paxos for forming a consistent sequence of append records.

In practice, we found that we needed to make many modifications and adaptations. In this keynote, we articulate the issues we encountered and the novel design solutions we employed to address them. In particular, we found that we needed to go beyond traditional state-machine-replication (SMR) and Paxos in more than one way:

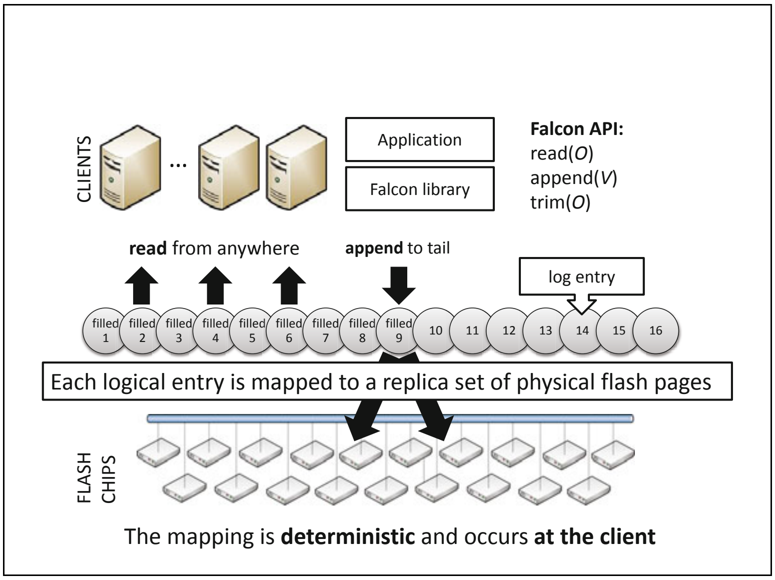


Fig. 1. The FALCON Design

– To allow parallel streaming of I/O, a conventional approach is to partition the storage space into autonomous partitions. In contrast, we partition the storage across **time**; each entry in the shared log is mapped onto a set of devices, thus allowing perfect load balance independent of the storage workload. In addition, this allows us to support higher semantics such as multi-object atomicity and transactions.
– The prevalent design in replicated storage systems is to employ primary-backup replication. This induces an I/O bottleneck at the primary, and limits throughput at the I/O capacity of a single server. Our design separates sequencing control from I/O, thus removing this bottleneck altogether. Though in itself, this idea is not new, we are aware of no previous system which employs this scheme.
– Each time we fill up the space in a set of storage drives, we shift to a new configuration. However, we continue to maintain old configurations in order to access earlier data in the log. Thus, our reconfiguration mechanism actively handles a list of configurations which map to a sequence of contiguous segments of the log. Accordingly, the read load of the shared log can be distributed over different sets of flash drives.
– In order to allow us to operate directly over passive storage devices, we need to adapt all protocols to a data centric model. The challenge in this model is that storage devices are not allowed to communicate with each other. Our solutions empower clients with most of the responsibility while employing thin servers.

Here we only briefly outline the Falcon design; the full architecture and implementation is described in detail elsewhere [Microsoft Technical Report MSR-TR-2011-86]. To append data to the shared log, a client first obtains a token from a *tokenserver* indicating the next free position in the shared log. The client then uses a *configuration* – basically, a membership view of the storage cluster – to deterministically map this token to a replica set of physical flash pages in the cluster. The client writes its data directly to these flash pages across the network. To read the entry at a specific position in the shared log, the client similarly uses the configuration to map the position to a set of physical flash pages, and then reads the data directly from one of these pages. When a drive fails in the cluster, clients in the system use a reconfiguration protocol to transition to a new configuration. Figure 1 gives a high level view of the system architecture.

The Falcon design has been fully implemented, demonstrating the feasibility of our design. The current Falcon implementation has been deployed over a cluster of 32 Intel X25M server-attached SSDs. This deployment currently supports 400K 4KB reads/sec and 200K 4KB appends/sec. Several applications have been prototyped over Falcon, including a transactional key-value store and a fully replicated database. While we are still evaluating these applications, the initial results are promising; for instance, our key-value store can support atomic multi-gets and multi-puts involving ten 4KB keys each at speeds of 40K/sec and 20K/sec, respectively.

Byzantizing Paxos by Refinement

Leslie Lamport

Microsoft Research

Abstract. We derive a $3f+1$ process Byzantine Paxos consensus algorithm by Byzantizing a variant of the ordinary Paxos algorithm—that is, by having $2f+1$ nonfaulty processes emulate the ordinary Paxos algorithm despite the presence of f malicious processes. We have written a formal, machine-checked proof that the Byzantized algorithm implements the ordinary Paxos consensus algorithm under a suitable refinement mapping.

You can verb anything.
Ron Ziegler (quoted by Brian Reid)

1 Introduction

The Paxos algorithm [6] has become a standard tool for implementing fault-tolerant distributed systems. It uses $2f + 1$ processes to tolerate the benign failure of any f of them. More recently, Castro and Liskov developed a $3f + 1$ process algorithm [2] that tolerates f Byzantine (maliciously faulty) processes. Intuitively, their algorithm seems to be a Byzantine version of Paxos. Other algorithms that also seem to be Byzantine versions of Paxos have subsequently appeared [4,11,14].

The only previous attempt we know of to explain the relation between a Byzantine Paxos algorithm and ordinary Paxos was by Lampson [13]. He derived both from an abstract, non-distributed algorithm. We take a more direct approach and derive a Byzantine Paxos algorithm from a distributed non-Byzantine one by a procedure we call *Byzantizing*, which converts an N process algorithm that tolerates the benign failure of up to f processes into an $N + f$ process algorithm that tolerates f Byzantine processes. In the Byzantized algorithm, the N good processes emulate the execution of the original algorithm despite the presence of f Byzantine ones. (Of course, a good process does not know which of the other processes are Byzantine.)

The heart of ordinary or Byzantine Paxos is a consensus algorithm. We Byzantize a variant of the classic Paxos consensus algorithm, which we call *PCon*, to obtain an abstract generalization of the Castro-Liskov Byzantine consensus algorithm that we call *BPCon*. (Section 3 explains why we do not Byzantize the original Paxos consensus algorithm.)

It is easy to make something appear simple by hand-waving. The fact that *BPCon* is derived from *PCon* is expressed formally by a TLA$^+$ [7] theorem asserting that *BPCon* implements *PCon* under a suitable refinement mapping [1].

(A derivation is an implementation proof presented backwards.) A formal proof of the safety part of this theorem has been written and checked by the TLAPS proof system; it is available on the Web [5]. We discuss liveness informally. We believe that other Byzantine Paxos consensus algorithms can also be derived by Byzantizing versions of Paxos, but we have not proved any other derivation.

We describe algorithms *PCon* and *BPCon* informally here. Their formal specifications are on the Web, along with the correctness proof [5]. (A pretty-printed version of the algorithms' PlusCal [9] code is also available on the Web site.) Section 8 explains just what this proof proves. In Section 7, we describe how the Castro-Liskov algorithm refines algorithm *BPCon*.

2 Consensus and Classic Paxos

We assume the usual distributed-computing model of asynchronous processes communicating by messages. By a benign failure, we mean the loss of a message or a process stopping. A Byzantine process may send any message, but we assume that the identity of the sender of a message can be determined by the receiver. This can be achieved by either point-to-point communication or message authenticators (MACs), which are described in Section 6.1.

2.1 Consensus

In a complete specification of consensus, *proposer* processes propose values, a set of *acceptor* processes together choose one of the proposed values, and *learner* processes learn what value, if any, has been chosen. The algorithm must tolerate the failure of some number f of acceptor processes, as well as the failure of any proposer or learner process.

To simplify the formal development, we eliminate the proposers and learners, and we consider only acceptors. Our definition of what value is chosen makes it clear how learning is implemented. Implementing proposers is not trivial in the Byzantine case, since one must prevent a Byzantine process from pretending to be a nonfaulty proposer. It becomes trivial by using digital signatures, and Castro and Liskov explain how it is done with MACs.

With this simplification, the specification of consensus consists of a trivial algorithm in which the acceptors can choose at most one value, but once chosen a value must remain forever chosen.

It is well-known that fault-tolerant consensus cannot be implemented in a purely asynchronous system [3]. We require that the safety properties (at most one value chosen and a value never unchosen) hold even in the absence of any synchrony assumption, and that liveness (a value is eventually chosen) holds under suitable synchrony assumptions on nonfaulty processes and the communication among them.

2.2 Paxos Consensus

The classic Paxos consensus algorithm was described in [6] and independently stated without proof by Oki [15]. It performs numbered ballots, each orchestrated by a leader. Multiple ballots may be performed concurrently (with different leaders). Once an acceptor performs an action in a ballot, it never performs any further actions of a lower-numbered ballot. We assume that ballots are numbered by natural numbers.

Let N be the number of acceptors, where $N > f$, and let a *quorum* be any $N-f$ acceptors. For safety, we require that any two quorums have a non-empty intersection, which is true if $N > 2f$. The only other property of quorums we use is that there is a quorum consisting entirely of nonfaulty processes, which is required for liveness.

An acceptor can *vote* for at most one value in any ballot. A value v is *chosen in* a ballot iff a quorum of acceptors have voted for v in that ballot. A value is *chosen* iff it is chosen in some ballot.

We say that a value v is *safe at* a ballot number b if no value other than v has been chosen or ever can be chosen in any ballot numbered less than b. (Although described intuitively in temporal terms, *safe at* is actually a function of the algorithm's current state.) The algorithm maintains the following properties:

P1. An acceptor can vote for a value v in ballot b only if v is safe at b.
P2. Different acceptors cannot vote for different values in the same ballot.

These properties are maintained by having the ballot-b leader choose a single value v that is safe at b and asking the acceptors to vote for v in ballot b. An acceptor will vote only when it receives such a request (and only if it has not performed any action of a higher-numbered ballot). A ballot b proceeds in two phases, with the following actions.

Phase 1a The ballot-b leader sends a $1a$ message to the acceptors.
Phase 1b An acceptor responds to the leader's ballot-b $1a$ message with a $1b$ message containing the number of the highest-numbered ballot in which it has voted and the value it voted for in that ballot, or saying that it has cast no votes.
Phase 2a Using the $1b$ messages sent by a quorum of acceptors, the leader chooses a value v that is safe at b and sends a $2a$ message containing v to the acceptors.
Phase 2b Upon receipt of the leader's ballot-b $2a$ message, an acceptor votes for v in ballot b by sending a $2b$ message.

(Remember that an acceptor performs a ballot-b Phase 1b or 2b action only if it has not performed an action for a higher-numbered ballot.) A value v is *chosen* iff a quorum of acceptors have voted for v in some ballot. A learner learns that a value has been chosen if it receives $2b$ messages from a quorum of acceptors for the same ballot (which by P2 must all report votes for the same value). However, since we are not modeling learners, the $2b$ messages serve only to record votes.

In its Phase 2a action, the ballot-b leader must determine a safe value from the ballot-b $1b$ messages it receives from a quorum. It does this by using the following properties of the algorithm.

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b.

P3b. If some acceptor in the quorum has voted, let c be the highest-numbered ballot less than b in which such a vote was cast. The value voted for in ballot c is safe at b. (By P2, there is only one such value.)

Paxos implements a state machine by executing an infinite sequence of separate instances of the consensus algorithm. There is normally a single leader executing ballots, using the same ballot number in all the instances. If that leader fails, a new leader executes Phase 1 for a higher-numbered ballot simultaneously for all instances of the consensus algorithm. For all instances in which a ballot was begun but learners may not know the chosen value, Phase 2 is executed immediately. For ballots not begun, in which P3a holds, the leader waits until it receives the necessary client proposals before executing Phase 2.

The ballot-b leader can always execute the Phase 1a action, and it can execute the Phase 2a action if it has received $1b$ messages from a quorum of acceptors. An acceptor can respond to messages from the leader if it has received no message from a higher-numbered ballot. Therefore, the ballot-b leader and a nonfaulty quorum of acceptors can choose a value if no higher-numbered ballot is begun. The liveness property satisfied by classic Paxos consensus is obtained directly from this observation; we will not bother stating it precisely. We just point out that the essential property from which liveness follows is the ability of the ballot-b leader to determine a safe value in Phase 2a from the ballot-b $1b$ messages sent by a quorum of acceptors.

3 Byzantizing an Algorithm

We Byzantize a consensus algorithm by having N acceptors emulate it in the presence of f *fake* acceptors—Byzantine processes that pretend to be acceptors. (Everything works with $m \leq f$ fake acceptors, but for simplicity we omit this generalization.) We sometimes call the acceptors *real* to more clearly distinguish them from the fake acceptors. Processes other than acceptors may be Byzantine—in particular, a Byzantized Paxos algorithm must tolerate malicious leaders. However, assumptions about the non-malicious behavior of leaders is required for liveness.

Formally, emulation means performing an action that, under a refinement mapping, is an action of the emulated algorithm. A refinement mapping maps each state of the emulating system (the implementation) to a state of the emulated one (the specification). Refinement mappings are explained in more detail in Section 8.

We are effectively assuming that which processes may be malicious are determined in advance. Since the Byzantized algorithm assumes no knowledge of which are the real acceptors and which the fake ones, this assumption results in no loss of generality. (It can be viewed as adding a prophecy variable [1] whose value always equals the set of processes that may fail.) Moreover, since a malicious process can do anything, including acting like a nonfaulty one, we can prove that the algorithm tolerates at least f malicious acceptors by assuming that there are exactly f fake acceptors that are malicious from the start.

We define the set of *byzacceptors* to be the union of the sets of real and fake acceptors. We define a *byzquorum* to be a set of byzacceptors that is guaranteed to contain a quorum of acceptors. If a quorum consists of any q acceptors, then a byzquorum consists of any $q+f$ byzacceptors. For liveness, we need the assumption that the set of all real acceptors (which we assume never fail) form a byzquorum.

In the Byzantized algorithm, a nonfaulty process must ensure that each action in its emulation is enabled by the original algorithm. For example, if we were modeling learners, the action of learning that a value v is chosen would be enabled by the receipt of ballot-b $2b$ messages with value v from a quorum of acceptors. In the Byzantized algorithm, the learner could perform that action when it had received such messages from a byzquorum, since that set of messages would contain a subset from a quorum of acceptors.

The key action in Paxos consensus is the leader's Phase 2a action, which chooses a safe value based on properties P3a and P3b. The leader can deduce that P3a holds if it receives $1b$ messages from a byzquorum, each asserting that the sender has not voted, because that byzquorum contains a quorum of acceptors. However, P3b is problematic. In the original algorithm, it is satisfied if there is a $1b$ message from some single acceptor reporting a vote in a ballot c. However, in the Byzantized algorithm, there is no way to determine if a single message is from a real or fake acceptor. One can maintain safety by requiring that a vote be reported in the highest-numbered ballot c by $f+1$ byzacceptors. However, liveness would then be lost because it is possible to reach a state in which this condition does not hold for the $1b$ messages sent by the real acceptors.

One way to fix this problem is to assume $N > 3f$. In that case, any two quorums have at least $f+1$ acceptors in common, and we can replace P3a and P3b by

- P3a$'$. If there is no ballot numbered less than b in which $f+1$ acceptors have voted, then all values are safe at b.
- P3b$'$. If there is some ballot c in which acceptors have voted and there is no higher-numbered ballot less than b in which $f+1$ acceptors have voted, then the value v voted for in c is safe at b.

The Phase 2a action is then always enabled by the receipt of $1b$ messages from a byzquorum because, if P3a$'$ does not hold, then we can apply P3b$'$ with c the largest ballot in which $f+1$ byzacceptors have voted for the same value. However, this is unsatisfactory because if assumes $N > 3f$, so it leads to a Byzantine consensus algorithm requiring more than $4f$ acceptors. Our solution to this problem is to use the variant of the Paxos consensus algorithm described in Section 4 below.

There is still another problem to be solved. For a Phase 2a action to be enabled, it is not enough for the leader to have received $1b$ messages from a quorum; it is also necessary that the leader has not already sent a (different) $2a$ message. If P3a holds, a malicious leader could send two $2a$ messages for different safe values. This could lead to two different values being chosen in two later ballots.

The solution to this problem lies in having the leader and the acceptors cooperatively emulate the execution of the Phase 2a action, using a new Phase 2av action. The leader sends to the byzacceptors a request to execute the Phase 2a action for a particular value v. An acceptor responds to this request by executing a Phase 2av action in which it sends a $2av$ message with value v to all the byzacceptors. It executes the Phase 2av action only if (i) it can determine that one such $2a$ message could be sent in the emulated algorithm (we explain in Section 5 how it does this), and (ii) it has not already executed a Phase 2av action in the current ballot. An acceptor can execute the Phase 2b action if it has received $2av$ messages with the same value from a byzquorum. Since any two byzquorums have a (real) acceptor in common, no two acceptors can execute Phase 2b actions for different values. The refinement mapping is defined so an emulated $2a$ message is considered to have been sent when a quorum of acceptors have sent the corresponding $2av$ messages.

4 Algorithm *PCon*

We now describe a variant called *PCon* of the classic Paxos consensus algorithm. call *PCon*. As explained below, a more general version of this algorithm has appeared before. Like classic Paxos, it assumes N acceptors with $N > 2f+1$.

In the classic algorithm described above, a ballot-b $2a$ message serves two functions: (i) it asserts that a value is safe at b, and (ii) it instructs the acceptors to vote for that value in ballot b. In algorithm *PCon*, we introduce a $1c$ message to accomplish (i), and we allow the leader to send multiple $1c$ messages asserting that multiple values are safe. We introduce a leader Phase 1c action and modify the Phase 2a action as follows:

Phase 1c. Using the $1b$ messages from a quorum of acceptors, the leader chooses a set of values that are safe at b and sends a $1c$ message for each of those values.

Phase 2a. The leader sends a $2a$ message for some value for which it has sent a $1c$ message.

The leader does not have to send all its $1c$ messages at once; it can execute the Phase 1c action multiple times in a single ballot. To choose safe values in the Phase 1c action, the ballot-b leader uses the following properties of the algorithm after receiving $1b$ messages from a quorum of acceptors.

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b.

P3c. If a ballot-c $1c$ message with value v has been sent, for some $c < b$, and (i) no acceptor in the quorum has voted in any ballot greater than c and less than b, and (ii) any acceptor in the quorum that has voted in ballot c voted for v in that ballot, then v is safe at b.

The careful reader will have noticed that we have not specified to whom the ballot-b leader sends its $1c$ messages, or how it learns about $1c$ messages sent in lower-numbered ballots so it can check if P3c holds. In algorithm *PCon*, the $1c$ messages are logical constructs that need not actually be sent. Sending a $2a$ message implies that the necessary $1c$ message was sent, and a $1b$ message reporting a vote in ballot c implies that a ballot-c $1c$ message was sent. So, why were $1c$ messages introduced in previous algorithms?

Systems that run for a long time cannot be based on a fixed set of acceptors. Acceptors must occasionally be removed and new ones added—a procedure called *reconfiguration*. In classic Paxos, reconfiguration happens between consensus instances, and a single instance is effectively executed by a single set of acceptors. Two algorithms have been proposed in which reconfiguration happens within the execution of a single consensus instance, with different ballots using possibly different sets of acceptors: Vertical Paxos [10] and an unpublished version of Cheap Paxos [12]. The $1c$ messages serve to eliminate the dependence on acceptors from lower-numbered ballots, which may have been reconfigured out of the system. When a new active leader begins ballot b, case P3a holds for the infinitely many instances for which Phase 2 of ballot b has not yet begun. The leader's $1c$ messages inform future leaders of this fact, so they do not have to learn about votes cast in any ballot numbered less than b.

The astute reader will have observed that the definition of *safe at* implies that if two different values are safe at b, then all values are safe at b. There is no reason for the leader to do anything other than sending a message saying a single value is safe, or sending messages saying that all values are safe. However, the more general algorithm is just as easy to prove correct and is simpler to Byzantize.

5 Algorithm *BPCon*

We now derive algorithm *BPCon* by Byzantizing the N-acceptor algorithm *PCon*, adding f fake acceptors. We first consider the actions of a leader process. There is no explicit $2a$ message or Phase 2a action in algorithm *BPCon*. Instead, the acceptors cooperate to emulate the sending of a $2a$ message, as described above in Section 3. The ballot-b leader requests that a Phase 2a action be performed for a value v for which it has already sent a $1c$ message. On receiving the first such request, an acceptor executes a Phase 2av action, sending a ballot-b $2av$ message for value v, if it has already received a legal ballot-b $1c$ message with that value.

Since the leader's request is necessary only for liveness, we do not explicitly model it. Instead, we allow an acceptor to perform a ballot-b Phase 2av action iff it has received the necessary $1c$ action and has not already sent a ballot-b $2av$ message.

Because the algorithm must tolerate malicious leaders, we let the ballot-b leader send any $1a$ and $1c$ messages it wants. (Remember that we assume a process cannot send a message that appears to be from another process.) There is only one possible ballot-b $1a$ message, and algorithm *PCon*'s Phase 1a action allows the leader to send it at any time. Hence the *BPCon* Phase 1a action is the same as the corresponding *PCon* action. The *BPCon* Phase 1c action allows the ballot-b leader to send any ballot-b $1c$ message at any time.

Acceptors will ignore a $1c$ message unless it is legal. To ensure liveness, a nonfaulty leader must send a message that (real) acceptors act upon. To see how it does that, we must determine how an acceptor knows that a $1c$ message is legal.

The sending of a ballot-b $1c$ message is enabled in *PCon* by P3a or P3c above, which requires the receipt of a set of $1b$ messages from a quorum and possibly of a $1c$ message. In *BPCon*, we put into the $1b$ messages additional information to enable the deduction that a $1c$ message was sent. An acceptor includes in its $1b$ messages the set of all $2av$ messages that it has sent—except that for each value v, it includes (and remembers) only the $2av$ message with the highest numbered ballot that it sent for v. Each of those $2av$ messages was sent in response to a legal $1c$ message. As explained in our discussion of Byzantizing in Section 3, this implies that given a set S of ballot-b $1b$ messages sent by a byzquorum, the following two conditions imply P3a and P3c, respectively:

BP3a. Each message in S asserts that its sender has not voted.

BP3c. For some $c < b$ and some value v, (a) each message in S asserts that (i) its sender has not voted in any ballot greater than c and (ii) if it voted in c then that vote was for v, and (b) there are $f+1$ $1b$ messages (not necessarily in S) from byzacceptors saying that they sent a $2av$ message with value v in ballot c.

A little thought shows we can weaken condition (b) of BP3c to assert:

(b′) there are $f+1$ $1b$ messages from byzacceptors saying that they sent a $2av$ message with value v in a ballot $\geq c$.

The c of P3c is then the largest of those ballot numbers $\geq c$ reported by a real acceptor.

To determine if a $1c$ message is legal, each acceptor maintains a set of $1b$ messages that it knows have been sent. Our abstract algorithm assumes an action that nondeterministically adds to that set any subset of $1b$ messages that have actually been sent. Of course, some of those $1b$ messages may be from fake acceptors, which may send any $1b$ message. Liveness requires the leader to ensure that the acceptors eventually know that the $1b$ messages enabling its sending of the $1c$ message have been sent. We discuss in Section 6 below how that is done.

As described above in Section 3, an acceptor performs a *Phase*$2b$ action when it knows that it has received identical $2av$ messages from a quorum of acceptors. A $2a$ message of *PCon* is emulated by a set of identical $2av$ messages sent by a quorum, with the Phase 2a action emulated by the sending of the last of that set of messages.

6 Liveness and Learning about Sent Messages

Liveness of *PCon* requires that a nonfaulty leader executes a ballot b, no leader begins a higher-numbered ballot, and the leader and nonfaulty acceptors can communicate with one another. The requirements for liveness of *BPCon* are the same. However, it is difficult to ensure that a Byzantine leader does not execute a higher-numbered ballot. Doing this seems to require an engineering solution based on real-time assumptions. One such solution is presented by Castro and Liskov.

Assuming these requirements, liveness of *BPCon* requires satisfying the following two conditions:

BL1. The leader can find $1b$ messages satisfying $BP3a$ or $BP3c$.
BL2. All real acceptors will know that those messages have been sent.

These two conditions imply that the leader will send a legal $1c$ message, a byzquorum BQ of real (nonfaulty) acceptors will receive that $1c$ message and send $2av$ messages, all the acceptors in BQ will receive those $2av$ messages and send $2b$ messages. Learners, upon receiving those $2b$ messages will learn that the value has been chosen.

To show that BL1 holds, observe that the ballot-b leader will eventually receive $1b$ messages from the acceptors in BQ. Let S be the set of those $1b$ messages. We now show that BP3a or BP3c holds.

1. It suffices to assume that BP3a is false and prove BP3c.
   Proof Obvious.
2. Let c be the largest ballot in which an acceptor in BQ voted, let a be such an acceptor, and let v be the value it voted for.
   Proof The existence of such a c follows from 1.
3. Acceptor a received ballot-c $2av$ messages with value v from a byzquorum.
   Proof By 2 and the enabling condition of the Phase 2b action.
4. No acceptor voted for a value other than v in ballot c.
   Proof By 3, since any two byzquorums have an acceptor in common and an acceptor can send at most one ballot-c av message.
5. At least $f+1$ acceptors sent ballot-c $2av$ messages with value v.
   Proof By 3, since a byzquorum contains at least $f+1$ acceptors.
6. Condition (b′) of BP3c holds.
   Proof By 5, because an acceptor sending a ballot-c $2av$ message with value v implies that, for $b > c$, its ballot-b $1b$ message will report that it sent a $2av$ message with value v in some ballot $\geq c$.
7. Condition (a) of BP3c holds.
   Proof By 2 (no acceptor in BQ voted in a ballot $> c$) and 4.
8. QED
   Proof By 1, 6, and 7.

This shows that BL1 eventually holds. To prove liveness, we need to show that BL2 holds. To ensure that it holds, the leader must have a way of ensuring that all the real acceptors eventually learn that a $1b$ message was sent. If the $1b$ message was sent by a real acceptor, then that acceptor can just broadcast its $1b$ message to all the byzacceptors as well as to the leader. We now present two methods for ensuring that an acceptor learns that a $1b$ message was sent, even if it was sent by a fake acceptor.

6.1 Sending Proofs

The simplest approach is for the leader to include with its $1c$ message a proof that all the necessary $1b$ messages have been sent. The easiest way to do that is to use full digital signatures and have byzacceptors sign their $1b$ messages. The leader can just include the necessary properly signed $1b$ messages in its $1c$ message.

There is another way for the leader to include in its $1c$ message a proof that a message was sent, using only authentication with MACs. A MAC is a signature $m_{p \rightarrow q}$ that a process p can attach to a message m that proves to q that p sent m. The MAC $m_{p \rightarrow q}$ proves nothing to any process other than q. We now describe a general method of obtaining a proof of a fact in the presence of f Byzantine processes. We can apply it to the fact that a process p sent a particular message.

Suppose a message m asserts a certain fact, and process q receives it with MAC $m_{p \rightarrow q}$ from $f+1$ different processes p. With at most f Byzantine processes, at least one of those processes p asserting the fact is nonfaulty, so the fact must be true. However, q cannot prove to any other process that the fact is true. However, suppose that it receives from $2f+1$ processes p the message together with a vector $\langle m_{p \rightarrow r_1}, \ldots, m_{p \rightarrow r_k} \rangle$ of MACs for the k processes $r_1, \ldots, r_k$. At least $f+1$ of those vectors were sent by nonfaulty processes p, so they have correct MACs and will therefore convince each r_i that a nonfaulty process sent m. Therefore, q can send m and these $2f+1$ vectors of MACs to each of the processes r_i as a proof of the fact asserted by m.

In general, a vector of $(j+1)f + 1$ MACs provides a proof that can be sent along a path of length j. For *BPCon*, we need it only for $j = 1$; one method of Byzantizing fast Paxos [8] uses the $j = 2$ case [11].

6.2 Relaying $1b$ Messages

We now describe another way a leader can ensure that good acceptors learn that a $1b$ message was sent. We have byzacceptors broadcast their $1b$ messages to all byzacceptors (as well as to the leader), and have them relay the $1b$ messages to the leader and to all other byzacceptors. Upon receipt of copies of a $1b$ message from $2f+1$ byzacceptors, the leader knows that at least $f+1$ real acceptors sent or relayed that message to all byzacceptors. Assuming the requirements for liveness, this implies that all acceptors will eventually receive copies of the $1b$ message from $f+1$ different byzacceptors, from which they infer that the message actually was sent.

This is the basic method used by Castro and Liskov. However, in their algorithm, the byzacceptors relay the broadcast $1b$ messages (which they call *view-change-acks*) only to the leader (which they call the *primary*). The leader includes (digests) of the $1b$ messages in its $1c$ message, and an acceptor asks the other byzacceptors to relay any $1b$ message that it hasn't received that is in the $1c$ message.

7 The Castro-Liskov Algorithm

The Castro-Liskov algorithm, like Paxos, executes a state machine by executing an unbounded sequence of instances of a consensus algorithm. It contains engineering optimizations for dealing with the sequence of instances—in particular, for garbage collecting old instances and for transferring state to repaired processes. We believe that those optimizations can be obtained by Byzantizing the corresponding optimizations for classic Paxos, but they are irrelevant to consensus. Some other optimizations, such as sending message digests instead of messages, are straightforward details that we ignore for simplicity.

When we ignore these details and consider only the consensus algorithm at the heart of the Castro-Liskov algorithm, we are left with an algorithm that refines *BPCon*. In the Castro-Liskov algorithm, byzacceptors are called *replicas*. The ballot-b leader is the replica called the *primary*, other byzacceptors being called *backups*. The replicas also serve as learners.

We explain how the Castro-Liskov consensus algorithm refines *BPCon* by describing how the messages of *BPCon* are implemented. We assume the reader is familiar with their algorithm.

- $1a$ There is no explicit $1a$ message; its sending is emulated cooperatively by the replicas when they decide to begin a view change.
- $1b$ This is the *view-change* message.
- $1c$ During a view change, the *new-view* message acts like $1c$ messages for all the consensus instances. For an instance in which the primary instructs the replicas to choose a specific value, it is a $1c$ message with that value. For all other instances, it is a set of $1c$ messages for all values. (Condition BP3a holds in those other instances.) The acceptors check the validity of these $1c$ messages simultaneously for all instances.
- $2av$ This is a backup's *prepare* message. The *pre-prepare* message of the primary serves as its $2av$ message and as the message (not modeled in *BPCon*) that requests a Phase 2a action.
- $2b$ This is the *commit* message.

As explained in Section 6.2, the Castro-Liskov algorithm's *view-change-ack* is used to relay $1b$ messages to the leader. Its *reply* message is sent by replicas serving as learners to inform the client of the chosen value.

We explained in Section 3 the difficulty in Byzantizing classic Paxos. Our inability to obtain the Castro-Liskov algorithm from classic Paxos is not a deficiency of Byzantizing; it is due to the fact that the algorithm does not refine classic Paxos—at least, not under any simple refinement mapping. In the Castro-Liskov consensus algorithm, a leader may be required to pre-prepare a value v even though no replica ever committed v in a previous view. This cannot happen in classic Paxos.

8 The Formal Specifications and Proof

All of our specifications and proofs are available on the Web [5]. Here, we give an overview of what we have done.

In addition to deriving *BPCon* from *PCon*, we also derive *PCon* as follows. We start with a simple specification *Consensus* of consensus. We refine *Consensus* by an algorithm *Voting*, a high-level non-distributed consensus algorithm that describes the ballots and voting that underlie *PCon*. We then obtain *PCon* by refining *Voting*.

All the specifications are written in PlusCal, a high-level algorithm language that superficially resembles a toy programming language [9]. A PlusCal algorithm is automatically translated into a TLA$^+$ specification. It is these TLA$^+$ specifications that we verify. The specifications of *BPCon* and *PCon* describe only the safety properties of the algorithms, so we are verifying only safety for them. For *Voting* and *Consensus*, we have written specifications of both safety and liveness.

Each step in the derivation of *BPCon* from *Consensus* is described formally by a refinement mapping. To explain what this means, we review refinement mappings as defined in [1]. (They are slightly different in TLA$^+$, which uses a single state space for all specifications.)

Let Σ_S denote the state space of a specification S, and let Σ_S^ω be the set of sequences of states in Σ_S. The specification S is a predicate on Σ_S^ω, where $S(\sigma)$ is true for a state sequence σ iff σ represents a behavior (possible execution) permitted by S.

A refinement of a specification S by a specification R is described by a mapping ϕ from Σ_R to Σ_S. We extend ϕ in the obvious way (pointwise) to a mapping from Σ_R^ω to Σ_S^ω. If F is a function on Σ_S or Σ_S^ω, we define $\overline{F}$ to be the function on Σ_R or Σ_R^ω, respectively, that equals $F \circ \phi$. We say that R *refines* (or *implements*) S under ϕ iff R implies $\overline{S}$. Thus, verifying correctness of the refinement means verifying the formula $R \Rightarrow \overline{S}$.

We have proved the correctness of the refinement of *PCon* by *BPCon*, and our proof has been completely checked by the TLAPS proof system, with two exceptions:

– A few trivial facts about finite sets are assumed without proof—for example, that a finite, non-empty set of integers contains a maximal element. We used TLC to check for errors in the TLA$^+$ formulas that state these assumptions.
– A handful of simple steps in the complete TLA$^+$ proof require temporal-logic reasoning. These steps, and their proofs, are identical for every TLA$^+$ refinement proof of safety properties. Since TLAPS does not yet handle temporal reasoning, proofs of these steps were omitted.

We have also written a complete proof that *Voting* refines *Consensus*, including the liveness properties. Most of the non-temporal steps of that proof have been checked by TLAPS; see [5] for details. We have checked our refinement of *Voting* by *PCon* with the TLC model checker, using a large enough model to be confident that there are no "coding" errors in our specifications. That, combined with our understanding of the algorithms, gives us confidence that this refinement is correct.

Mathematical correctness of a refinement tells us nothing useful unless the refinement mapping is a useful one. For example, a simple counter implements

PCon under a refinement mapping in which the counter changing from n to $n+1$ is mapped to the execution of the Phase 1a action by the ballot-n leader. Our refinement mappings are intuitively reasonable, as indicated by our informal description of the refinement mapping for the refinement of *PCon* by *BPCon*. Because the purpose of a consensus algorithm is to determine what value is chosen, we can provide the following more rigorous demonstration that our refinement mappings are the "right" ones.

For each of our specifications, we define a state function *chosen* that equals the set of values that have been chosen (a set containing at most one value). Let $chosen_S$ be the state function *chosen* defined for specification S. The following relations among these state functions show that our refinement mappings are the ones we want.

$$\begin{aligned} chosen_{Voting} &= \overline{chosen_{Consensus}} \\ chosen_{PCon} &= \overline{chosen_{Voting}} \\ chosen_{BPCon} &\Rightarrow \overline{chosen_{PCon}} \end{aligned}$$

The last relation is implication rather than equality for the following reason. A value v is in $\overline{chosen_{PCon}}$ iff a quorum of acceptors have voted for v in the same ballot. It is impossible for a learner to determine if that is true because it does not know which byzacceptors are acceptors. We therefore define $chosen_{BPCon}$ to contain a value v iff a byzquorum has voted for it in the same ballot, which implies that v is in $\overline{chosen_{PCon}}$; hence the implication. (For liveness, we must show that any element of $\overline{chosen_{PCon}}$ is eventually an element of $chosen_{BPCon}$).

The first of these relations is essentially the definition of the refinement mapping under which *Voting* refines *Consensus*. The second has been checked by TLC. The third has been proved and the proof checked by TLAPS.

9 Conclusion

For a number of years, we have been informally explaining Byzantine consensus algorithms as the Byzantizing of ordinary Paxos. We decided that formalizing the Byzantizing of Paxos would be interesting in itself and would provide a test of how well TLAPS works on real problems.

Although the basic idea of Byzantizing was right, the formalization revealed that we were quite wrong in the details. In particular, we originally thought that the Castro-Liskov algorithm refined classic Paxos consensus. We wrote and checked almost the complete proof of that refinement, discovering the error only because we were unable to prove one of the last remaining steps. We are not sure if we would have found the error had we written a careful, hierarchically structured hand proof. We are quite sure that we would not have found it by writing a conventional paragraph-style "mathematical" proof.

Our proof that *BPCon* refines *PCon* revealed a number of problems with TLAPS that were then corrected. Our subsequent proof that *Voting* refines *Consensus* went quite smoothly. We intend to finish checking that proof when TLAPS supports temporal reasoning. We hope to prove that *BPCon* also refines

PCon when suitable liveness properties are added to the specifications. It would be nice to prove that *PCon* refines *Voting*. However, we probably won't bother because we are already convinced that the refinement is correct, and because its simpler proof would be less of a test of TLAPS.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
2. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
3. Fischer, M.J., Lynch, N., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
4. Guerraoui, R., Vukolić, M.: Refined quorum systems. Distributed Computing 23(1), 1–42 (2010)
5. Lamport, L.: Mechanically checked safety proof of a byzantine paxos algorithm, `http://research.microsoft.com/users/lamport/tla/byzpaxos.html`. The page can also be found by searching the Web for the 23-letter string obtained by removing the "-" from, `uid-lamportbyzpaxosproof`
6. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems 16(2), 133–169 (1998)
7. Lamport, L.: Specifying Systems. Addison-Wesley, Boston (2003)
8. Lamport, L.: Fast paxos. Distributed Computing 19(2), 79–103 (2006)
9. Lamport, L.: The pluscal algorithm language. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 36–60. Springer, Heidelberg (2009)
10. Lamport, L., Malkhi, D., Zhou, L.: Vertical paxos and primary-backup replication. In: Tirthapura, S., Alvisi, L. (eds.) Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, pp. 312–313. ACM, New York (2009)
11. Lamport, L.B.: Fast byzantine paxos. United States Patent 7620680, filed (August 15, 2002) issued (November 17, 2009)
12. Lamport, L.B., Massa, M.T.: Cheap Paxos. United States Patent 7249280, filed (June 18, 2004) issued (July 24, 2007)
13. Lampson, B.W.: The ABCDs of Paxos, `http://research.microsoft.com/lampson/65-ABCDPaxos/Abstract.html`
14. Martin, J.-P., Alvisi, L.: Fast byzantine consensus. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005), Yokohama, pp. 402–411. IEEE Computer Society, Los Alamitos (2006)
15. Oki, B.M.: Viewstamped replication for highly available distributed systems. Technical Report MIT/LCS/TR-423, MIT Laboratory for Computer Science (Ph.D. Thesis) (August 1988)

Unbounded Contention Resolution in Multiple-Access Channels*

Antonio Fernández Anta[1],
Miguel A. Mosteiro[2,3], and Jorge Ramón Muñoz[3]

[1] Institute IMDEA Networks, Madrid, Spain
antonio.fernandez@imdea.org
[2] Department of Computer Science, Rutgers University, Piscataway, NJ, USA
mosteiro@cs.rutgers.edu
[3] LADyR, GSyC, Universidad Rey Juan Carlos, Móstoles, Spain
jorge.ramon@madrimasd.net

Abstract. A frequent problem in settings where a unique resource must be shared among users is how to resolve the contention that arises when all of them must use it, but the resource allows only for one user each time. The application of efficient solutions for this problem spans a myriad of settings such as radio communication networks or databases. For the case where the number of users is unknown, recent work has yielded fruitful results for local area networks and radio networks, although either a (possibly loose) upper bound on the number of users needs to be known [7], or the solution is suboptimal [2], or it is only implicit [11] or embedded [6] in other problems, with bounds proved only asymptotically. In this paper, under the assumption that collision detection or information on the number of contenders is not available, we present a novel protocol for contention resolution in radio networks, and we recreate a protocol previously used for other problems [11, 6], tailoring the constants for our needs. In contrast with previous work, both protocols are proved to be optimal up to a small constant factor and with high probability for big enough number of contenders. Additionally, the protocols are evaluated and contrasted with the previous work by extensive simulations. The evaluation shows that the complexity bounds obtained by the analysis are rather tight, and that both protocols proposed have small and predictable complexity for many system sizes (unlike previous proposals).

1 Introduction

The topic of this work is the resolution of contention in settings where an unknown number of users must access a single shared resource, but multiple simultaneous accesses are not feasible. The scope of interest in this problem is

* A brief announcement of this work has been presented at PODC 2011. This work is supported in part by the Comunidad de Madrid grant S2009TIC-1692, Spanish MICINN grant TIN2008–06735-C02-01, and National Science Foundation grant CCF-0937829.

wide, ranging from radio and local area networks to databases and transactional memory. (See [2] and the references therein.)

A common theme in protocols used for this problem is the adaptive adjustment of some user variable that reflects its eagerness in trying to access the shared resource. Examples of such variable are the probability of transmitting a message in a radio network or the frequency of packet transmission in a local area network. When such adjustment reduces (resp. increases) the contention, the technique is called *back-off* (resp. *back-on*). Combination of both methods are called *back-on/back-off.* Protocols used may be further characterized by the rate of adjustment. E.g., *exponential back-off*, *polynomial back-on*, etc. In particular, exponential back-off is widely used and it has proven to be efficient in practical applications where statistical arrival of contenders is expected. Nevertheless, worst case arrival patterns, such as bursty or *batched* arrivals, are frequent [18,12].

A technique called Loglog-iterated Back-off was shown to be within a sublogarithmic factor from optimal with high probability in [2]. [1] The protocol was presented in the context of packet contention resolution in local area networks for batched arrivals. Later on, also for batched arrivals, we presented a back-on/back-off protocol in [7], instantiated in the k-selection problem in Radio Networks (defined in Section 2). The latter protocol, named here Log-Fails Adaptive, is asymptotically optimal for any significant probability of error, but additionally requires that some upper bound (possibly loose) on the number of contenders is known. In the present paper, we remove such requirement. In particular, we present and analyze a protocol that we call One-Fail Adaptive for k-selection in Radio Networks. We also recreate and analyze another protocol for k-selection, called here Exp Back-on/Back-off , which was previously embedded in protocols for other problems and analyzed only asymptotically [11,6]. Our analysis shows that One-Fail Adaptive and Exp Back-on/Back-off, both of independent interest, resolve contention among an unknown and unbounded[2] number of contenders with high probability in optimal time up to constants. Additionally, by means of simulations, we evaluate and contrast the average performance of all four protocols. The simulations show that the complexity bounds obtained in the analysis (with high probability) for these protocols are rather tight for the input sizes considered. Additionally, they show that they are faster that Loglog-iterated Back-off and more predictable for all network sizes than Log-Fails Adaptive.

Roadmap: The rest of the paper is organized as follows. In the following section the problem, model, related work and results are detailed. In Section 3, we introduce One-Fail Adaptive and its analysis. Exp Back-on/Back-off is detailed and analyzed in Section 4. The results of the empirical contrast of all

[1] For k contenders, we define *with high probability* to mean with probability at least $1 - 1/k^c$ for some constant $c > 0$.

[2] We use the term unbounded to reflect that not even an upper bound on the number of contenders is known. This should not be confused with the infinitely-many users model where there are countably infinitely many stations. [4]

four protocols is given in Section 5 and we finish with concluding remarks and open problems in Section 6.

2 Preliminaries

A well-studied example of unique-resource contention is the problem of broadcasting information in a multiple-access channel. A multiple-access channel is a synchronous system that allows a message to be delivered to many recipients at the same time using a channel of communication but, due to the shared nature of the channel, the simultaneous introduction of messages from multiple sources produce a conflict that precludes any message from being delivered to any recipient. The particular model of multiple-access channel we consider here is the Radio Network, a model of communication network where the channel is contended (even if radio communication is not actually used [4]). We first precise our model of Radio Network as follows.

The Model: We consider a Radio Network comprised of n stations called *nodes*. Each node is assumed to be potentially reachable from any other node in one communication step, hence, the network is characterized as *single-hop* or *one-hop* indistinctively. Before running the protocol, nodes have no information, not even the number of nodes n or their own label. Time is supposed to be slotted in *communication steps*. Assuming that the computation time-cost is negligible in comparison with the communication time-cost, time efficiency is studied in terms of communication steps only. The piece of information assigned to a node in order to deliver it to other nodes is called a *message*. The assignment of a message is due to an external agent and such an event is called a *message arrival*. Communication among nodes is carried out by means of radio broadcast on a shared channel. If exactly one node transmits at a communication step, such a transmission is called *successful* or *non-colliding*, we say that the message was *delivered*, and all other nodes *receive* such a message. If more than one message is transmitted at the same time, a *collision* occurs, the messages are garbled, and nodes only receive *interference noise*. If no message is transmitted in a communication step, nodes receive only *background noise*. In this work, nodes can not distinguish between interference noise and background noise, thus, the channel is called *without collision detection*. Each node is in one of two states, *active* if it holds a message to deliver, or *idle* otherwise. As in [2,16,11], we assume that a node becomes idle upon delivering its message, for instance when an explicit acknowledgement is received (like in the IEEE 802.11 Medium Access Control protocol [1]). For settings where the channel does not provide such functionality, such as Sensor Networks, a hierarchical infrastructure may be predefined to achieve it [6], or a leader can be elected as the node responsible for acknowledging successful transmissions [22].

One of the problems that require contention resolution in Radio Networks is the problem known in the literature as *all-broadcast* [4], or *k-selection* [16]. In k-selection, a set of k out of n network nodes have to access a unique shared channel of communication, each of them at least once. As in [2,16,11], in this paper we study k-selection when all messages arrive simultaneously, or in a *batch*. Under this assumption the k-selection problem is called *static*. A *dynamic* counterpart where messages arrive at different times was also studied [16].

The Problem: Given a Radio Network where k network nodes are activated by a message that arrives simultaneously to all of them, the *static k-selection* problem is solved when each node has delivered its message.

Related Work: A number of fruitful results for contention resolution have been obtained assuming availability of collision detection. Martel presented in [19] a randomized adaptive protocol for k-Selection that works in $O(k + \log n)$ time in expectation[3]. As argued by Kowalski in [16], this protocol can be improved to $O(k + \log \log n)$ in expectation using Willard's expected $O(\log \log n)$ selection protocol of [23]. In the same paper, Willard shows that, for any given protocol, there exists a choice of $k \leq n$ such that selection takes $\Omega(\log \log n)$ expected time for the class of fair selection protocols (i.e., protocols where all nodes use the same probability of transmission to transmit in any given time slot). For the case in which n is not known, in the same paper a $O(\log \log k)$ expected time selection protocol is described, again, making use of collision detection. If collision detection is not available, using the techniques of Kushilevitz and Mansour in [17], it can be shown that, for any given protocol, there exists a choice of $k \leq n$ such that $\Omega(\log n)$ is a lower bound in the expected time to get even the first message delivered.

Regarding deterministic solutions, the k-Selection problem was shown to be in $O(k \log(n/k))$ already in the 70's by giving adaptive protocols that make use of collision detection [3,13,20]. In all these results the algorithmic technique, known as *tree algorithms*, relies on modeling the protocol as a complete binary tree where the messages are placed at the leaves. Later, Greenberg and Winograd [10] showed a lower bound for that class of protocols of $\Omega(k \log_k n)$. Regarding oblivious algorithms, Komlòs and Greenberg [15] showed the existence of $O(k \log(n/k))$ solutions even without collision detection but requiring knowledge of k and n. More recently, Clementi, Monti, and Silvestri [5] showed a lower bound of $\Omega(k \log(n/k))$, which also holds for adaptive algorithms if collision detection is not available. In [16], Kowalski presented the construction of an oblivious deterministic protocol that, using the explicit selectors of Indyk [14], gives a $O(k \operatorname{polylog} n)$ upper bound without collision detection.

In [9], Gerèb-Graus and Tsantilas presented an algorithm that solves the problem of realizing arbitrary h-relations in an n-node network, with probability at least $1 - 1/n^c, c > 0$, in $\Theta(h + \log n \log \log n)$ steps. In an h-relation, each processor is the source as well as the destination of at most h messages. Making $h = k$ this protocol can be used to solve static k-selection. However, it requires that nodes know h.

Extending previous work on tree algorithms, Greenberg and Leiserson [11] presented randomized routing strategies in fat-trees for bounded number of

[3] Througout this paper, log means $\log_2$ unless otherwise stated.

messages. Underlying their algorithm lies a sawtooth technique used to "guess" the appropriate value for some critical parameter (load factor), that can be used to "guess" the number of contenders in static k-selection. Furthermore, modulo choosing the appropriate constants, Exp Back-on/Back-off uses the same sawtooth technique. Their algorithm uses n and it is analyzed only asymptotically.

Monotonic back-off strategies for contention resolution of batched arrivals of k packets on simple multiple access channels, a problem that can be seen as static k-selection, have been analyzed in [2]. In that paper, it is shown that r-*exponential back-off*, a monotonic technique used widely that has proven to be efficient for many practical applications is in $\Theta(k \log^{\log r} k)$ for batched arrivals. The best strategy shown is the so-called *loglog-iterated back-off* with a makespan in $\Theta(k \log\log k / \log\log\log k)$ with probability at least $1-1/k^c, c > 0$, which does not use any knowledge of k or n. In the same paper, the sawtooth technique used in [11] is informally described in a paragraph while pointing out that it yields linear time for contention resolution thanks to non-monotonicity, but no analysis is provided.

Later on, Farach-Colton and Mosteiro presented an optimal protocol for Gossiping in Radio Networks in [6]. The sawtooth technique embedded in [11] is used in that paper as a subroutine to resolve contention in linear time as in Exp Back-on/Back-off. However, the algorithm makes use of n to achieve the desired probability of success and the analysis is only asymptotical.

A randomized adaptive protocol for static k-selection in a one-hop Radio Network without collision detection was presented in [7]. The protocol is shown to solve the problem in $(e + 1 + \xi)k + O(\log^2(1/\varepsilon))$ steps with probability at least $1 - 2\varepsilon$, where $\xi > 0$ is an arbitrarily small constant and $0 < \varepsilon \leq 1/(n+1)$. Modulo a constant factor, the protocol is optimal if $\varepsilon \in \Omega(2^{-\sqrt{n}})$. However, the algorithm makes use of the value of ε, which must be upper bounded as above in order to guarantee the running time. Therefore, knowledge of n is required.

Our Results: In this paper, we present a novel randomized protocol for static k-selection in a one-hop Radio Network, and we recreate a previously used technique suiting the constants for our purpose and without making use of n. Both protocols work without collision detection and do not require information about the number of contenders. As mentioned, these protocols are called One-Fail Adaptive and Exp Back-on/Back-off. It is proved that One-Fail Adaptive solves static k-selection within $2(\delta+1)k+O(\log^2 k)$ steps, with probability at least $1-2/(1+k)$, for $\mathrm{e} < \delta \leq \sum_{j=1}^{5}(5/6)^j$. On the other hand, Exp Back-on/Back-off is shown to solve static k-selection within $4(1+1/\delta)k$ steps with probability at least $1 - 1/k^c$ for some constant $c > 0$, $0 < \delta < 1/\mathrm{e}$, and big enough k. Given that k is a lower bound for this problem, both protocols are optimal (modulo a small constant factor) for big enough number of contenders.

Observe that the bounds and the probabilities obtained are given as functions of the parameter k, as done in [2], since this is the input parameter of our version of the problem. A fair comparison with the results obtained as function of k and n would require that k is large enough, so that $n = \Omega(k^c)$, for some constant c.

Both protocols presented are of interest because, although protocol Exp Back-on/Back-off is simpler, One-Fail Adaptive achieves a better multiplicative factor, although the constant in the sublinear additive factor may be big for small values of k.

Additionally, results of the evaluation by simulation of the average behavior of One-Fail Adaptive and Exp Back-on/Back-off and a comparison with Log-Fails Adaptive and Loglog-iterated Back-off are presented. Both algorithms One-Fail Adaptive and Exp Back-on/Back-off run faster than Loglog-iterated Back-off on average, even for small values of k. Although Loglog-iterated Back-off has higher asymptotic complexity, one may have expected that it may run fast for small networks. On the other hand, the knowledge on a bound of k assumed by Log-Fails Adaptive seems to provide an edge with respect to One-Fail Adaptive and Exp Back-on/Back-off for large values of k. However, Log-Fails Adaptive has a much worse behavior than the proposed protocols for small to moderate network sizes ($k \leq 10^5$). In any case, for all values of k simulated, One-Fail Adaptive and Exp Back-on/Back-off have a very stable and efficient behavior.

3 One-Fail Adaptive

As in Log-Fails Adaptive [7], One-Fail Adaptive is composed by two interleaved randomized algorithms, each intended to handle the communication for different levels of contention. One of the algorithms, which we call *AT*, is intended for delivering messages while the number of nodes contending for the channel is above some logarithmic threshold (to be defined later). The other algorithm, called *BT*, has the purpose of handling message delivery after that number is below that threshold. Nonetheless, a node may transmit using the BT (resp. AT) algorithm even if the number of messages left to deliver is above (resp. below) that threshold.

Both algorithms, AT and BT, are based on transmission trials with certain probability and what distinguishes them is just the specific probability value used. It is precisely the particular values of probability used in each algorithm what differentiates One-Fail Adaptive from Log-Fails Adaptive. For the BT algorithm, the probability of transmission is inversely logarithmic on the number of messages already transmitted, while in Log-Fails Adaptive that probability was fixed. For the AT algorithm the probability of transmission is the inverse of an estimation on the number of messages left to deliver. In One-Fail Adaptive this estimation is updated continuously, whereas in Log-Fails Adaptive it was updated after some steps without communication. These changes yield a protocol still linear, but now it is not necessary to know n. Further details can be seen in Algorithm 1.

For clarity, Algorithms AT and BT are analyzed separately taking into account in both analyses the presence of the other. We show the efficiency of the AT algorithm in producing successful transmissions while the number of messages left is above some logarithmic threshold, and the efficiency of the BT algorithm

Algorithm 1. One-Fail Adaptive. Pseudocode for node x. δ is a constant such that $e < \delta \leq \sum_{j=1}^{5}(5/6)^j$.

```
upon message arrival do
    $\widetilde{\kappa} \leftarrow \delta + 1$                              // Density estimator
    $\sigma \leftarrow 0$                                  // Messages-received counter
    start tasks $1, 2$ and $3$
Task 1
    foreach communication-step $= 1, 2, \ldots$ do
        if communication-step $\equiv 0 \pmod 2$ then                     // BT-step
            transmit $\langle x, \text{message}\rangle$ with prob $1/(1 + \log(\sigma + 1))$
        else                                                            // AT-step
            transmit $\langle x, \text{message}\rangle$ with probability $1/\widetilde{\kappa}$
            $\widetilde{\kappa} \leftarrow \widetilde{\kappa} + 1$
Task 2
    upon reception from other node do
        $\sigma \leftarrow \sigma + 1$
        if communication-step $\equiv 0 \pmod 2$ then                     // BT-step
            $\widetilde{\kappa} \leftarrow \max\{\widetilde{\kappa} - \delta, \delta + 1\}$
        else                                                            // AT-step
            $\widetilde{\kappa} \leftarrow \max\{\widetilde{\kappa} - \delta - 1, \delta + 1\}$
Task 3
    upon message delivery stop
```

handling the communication after that threshold is crossed. For the latter, we use standard probability computations to show our time upper bound. For the AT algorithm, we use concentration bounds to show that the messages are delivered with large enough probability, while the density estimator $\widetilde{\kappa}$ does not exceed the actual number of messages left. This second proof is more involved since it requires some preliminary lemmas. We establish here the main theorem, which is direct consequence of the lemmata described that can be found in [8].

Theorem 1. *For any $e < \delta \leq \sum_{j=1}^{5}(5/6)^j$ and for any one-hop Radio Network under the model detailed in Section 1,* One-Fail Adaptive *solves static k-selection within $2(\delta + 1)k + O(\log^2 k)$ communication steps, with probability at least $1 - 2/(1 + k)$.*

4 Exp Back-on/Back-off

The algorithm presented in this section is based in contention windows. That is, each node repeatedly chooses uniformly one time slot within an interval, or *window*, of time slots to transmit its message. Regarding the size of such window, our protocol follows a back-on/back-off strategy. Namely, the window is increased in an outer loop and decreased in an inner loop, as detailed in Algorithm 2.

The intuition for the algorithm is as follows. Let m be the number of messages left at a given time right before using a window of size w. We can think of

Algorithm 2. Window size adjustment in Exp Back-on/Back-off. $0 < \delta < 1/\mathrm{e}$ is a constant.

```
1 for $i = \{1, 2, \dots\}$ do
2     $w \leftarrow 2^i$
3     while $w \geq 1$ do
4         Choose uniformly a step within the next $w$ steps
5         $w \leftarrow w \cdot (1 - \delta)$
```

the algorithm as a random process where m balls (modelling the messages) are dropped uniformly in w bins (modelling time slots). We will show that, if $m \leq w$, for large enough m, with high probability, at least a constant fraction of the balls fall alone in a bin. Now, we can repeat the process removing this constant fraction of balls and bins until all balls have fallen alone. Since nodes do not know m, the outer loop increasing the size of the window is necessary. The analysis follows.

Lemma 1. *For $k \geq m \geq (2\mathrm{e}/(1-\mathrm{e}\delta)^2)(1+(\beta+1/2)\ln k)$, $0 < \delta < 1/\mathrm{e}$, $m \leq w$, and $\beta > 0$, if m balls are dropped in w bins uniformly at random, the probability that the number of bins with exactly one ball is less than δm is at most $1/k^\beta$.*

Proof. Since a bigger number of bins can only reduce the number of bins with more than one ball, if the claim holds for $w = m$ it also holds for $w > m$. Thus, it is enough to prove the first case. The probability for a given ball to fall alone in a given bin is $(1/m)(1 - 1/m)^{m-1} \geq 1/(\mathrm{e}m)$. Let X_i be a random variable that indicates if there is exactly one ball in bin i. Then, $Pr(X_i = 1) \geq 1/\mathrm{e}$. To handle the dependencies that arise in balls and bins problems, we approximate the joint distribution of the number of balls in all bins by assuming the load in each bin is an independent Poisson random variable with mean 1. Let X be a random variable that indicates the total number of bins with exactly one ball. Then, $\mu = E[X] = m/\mathrm{e}$. Using Chernoff-Hoeffding bounds [21], $Pr(X \leq \delta m) \leq \exp\left(-m\,(1-\mathrm{e}\delta)^2/(2\mathrm{e})\right)$, because $0 < \delta < 1/\mathrm{e}$.

As shown in [21], any event that takes place with probability p in the Poisson case takes place with probability at most $p\mathrm{e}\sqrt{m}$ in the exact case. Then, we want to show that $\exp\left(-m(1-\mathrm{e}\delta)^2/(2\mathrm{e})\right)\mathrm{e}\sqrt{m} \leq k^{-\beta}$, which is true for $m \geq \frac{2\mathrm{e}}{(1-\mathrm{e}\delta)^2}\left(1 + \left(\frac{1}{2} + \beta\right)\ln k\right)$. □

Theorem 2. *For any constant $0 < \delta < 1/\mathrm{e}$, Exp Back-on/Back-off solves static k-selection within $4(1 + 1/\delta)k$ steps with probability at least $1 - 1/k^c$, for some constant $c > 0$ and big enough k.*

Proof. Consider an execution of the algorithm on k nodes. Let a round be the sequence of time steps corresponding to one iteration of the inner loop of Algorithm 2, i.e. the time steps of a window. Let a phase be the sequence of rounds corresponding to one iteration of the outer loop of Algorithm 2, i.e. when the window is monotonically reduced.

Consider the first round when $k \leq w < 2k$. Assume no message was transmitted successfully before. (Any messages transmitted could only reduce the running time.) By Lemma 1, we know that, for $0 < \delta < 1/\mathrm{e}$ and $\beta > 0$, at least δk messages are transmitted in this round with probability at least $1 - 1/k^\beta$, as long as $k \geq \tau$, where $\tau \triangleq (2\mathrm{e}/(1-\mathrm{e}\delta)^2)(1 + (\beta + 1/2)\ln k)$.

Conditioned on this event, for some $\delta_1 \geq \delta$ fraction of messages transmitted in the first round, using the same lemma we know that in the following round at least $\delta(1-\delta_1)k$ messages are transmitted with probability at least $1 - 1/k^\beta$, as long as $(1-\delta_1)k \geq \tau$. This argument can be repeated for each subsequent round until the number of messages left to be transmitted is less than τ. Furthermore, given that the size of the window is monotonically reduced within a phase until $w = 1$, even if the fraction of messages transmitted in each round is just δ, the overall probability of reducing the number of messages left from k to τ within this phase is at least $(1 - 1/k^\beta)^{\log_{1/(1-\delta)}(2k)}$.

Consider now the first round of the following phase, i.e. when $2k \leq w < 4k$. Assume that at most τ nodes still hold a message to be transmitted. Using the union bound, the probability that two or more of m nodes choose a given step in a window of size w is at most $\binom{m}{2}/w^2$. Applying again the union bound, the probability that in any step two or more nodes choose to transmit is at most $\binom{m}{2}/w \leq \binom{\tau}{2}/(2k) = \tau(\tau+1)/(4k)$.

Therefore, using conditional probability, in order to complete the proof, it is enough to show that

$$\left(1 - \frac{\tau(\tau+1)}{4k}\right)\left(1 - \frac{1}{k^\beta}\right)^{\log_{1/(1-\delta)}(2k)} \geq 1 - \frac{1}{k^c}, \text{ for some constant } c > 0$$

$$\exp\left(-\frac{\tau(\tau+1)}{4k - \tau(\tau+1)} - \frac{\log_{1/(1-\delta)}(2k)}{k^\beta - 1}\right) \geq \exp\left(-\frac{1}{k^c}\right)$$

$$\frac{\tau(\tau+1)}{4k - \tau(\tau+1)} + \frac{\log_{1/(1-\delta)}(2k)}{k^\beta - 1} \leq \frac{1}{k^c}. \tag{1}$$

Given that δ is a constant and fixing $\beta > 0$ as a constant, Inequality 1 is true for some constant $c < \min\{1, \beta\}$, for big enough k. Telescoping the number of steps up to the first round when $w = 4k$, the running time is less than $4k + 2k\sum_{i=0}^{\infty}\sum_{j=0}^{\infty}(1-\delta)^j/2^i = 4(1 + 1/\delta)k$. □

5 Evaluation

In order to evaluate the expected behavior of the algorithms One-Fail Adaptive and Exp Back-on/Back-off, and compare it with the previously proposed algorithms Loglog-iterated Back-off and Log-Fails Adaptive, we have simulated the four algorithms. The simulations measure the number of steps that the algorithms take until the static k-selection problem has been solved, i.e., each of the k activated nodes of the Radio Network has delivered its message, for different values of k. Several of the algorithms have parameters that can be adapted. The value of these parameters is the same for all the simulations of

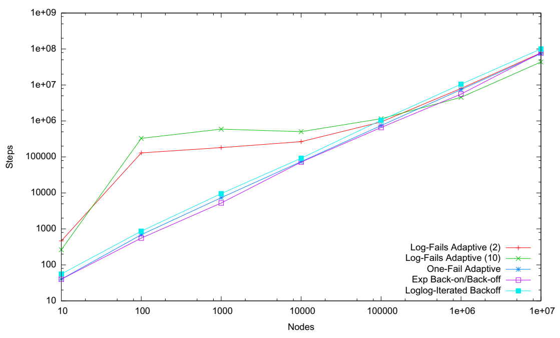


Fig. 1. Number of steps to solve static k-selection, per number of nodes k

Table 1. Ratio steps/nodes as a function of the number of nodes k

k	10	10^2	10^3	10^4	10^5	10^6	10^7	Analysis
Log-Fails Adaptive $\xi_t = 1/2$	46.4	1292.4	181.9	26.6	9.4	8.0	7.8	7.8
Log-Fails Adaptive $\xi_t = 1/10$	26.3	3289.2	593.8	50.3	11.5	4.5	4.4	4.4
One-Fail Adaptive	4.0	6.9	7.4	7.4	7.4	7.4	7.4	7.4
Exp Back-on/Back-off	4.0	5.5	5.2	7.2	6.6	5.6	7.9	14.9
Loglog-iterated Back-off	5.6	8.6	9.6	9.2	10.5	10.5	10.1	$\Theta\left(\frac{\log\log k}{\log\log\log k}\right)$

the same algorithm (except the parameter ε of Log-Fails Adaptive that has to depend on k). For Exp Back-on/Back-off the parameter is chosen to be $\delta = 0.366$. For One-Fail Adaptive the parameter is chosen to be $\delta = 2.72$. For Log-Fails Adaptive, the parameters (see their meaning in [7]) are chosen to be $\xi_\delta = \xi_\beta = 0.1$ and $\varepsilon \approx 1/(k+1)$, while two values of ξ_t have been used, $\xi_t = 1/2$ and $\xi_t = 1/10$. Finally, Loglog-iterated Back-off is simulated with parameter $r = 2$ (see [2]).

Figure 1 presents the average number of steps taken by the simulation of the algorithms. The plot shows the the average of 10 runs for each algorithm as a function of k. In this figure it can be observed that Log-Fails Adaptive takes significantly larger number of steps than the other algorithms for moderately small values of k (up to 10^5). Beyond $k = 10^5$ all algorithms seem to have a similar behavior.

A higher level of detail can be obtained by observing Table 1, which presents the ratio obtained by dividing the number of steps (plotted in Figure 1) by the value of k, for each k and each algorithm. In this table, the bad behavior of Log-Fails Adaptive for moderate values of k can be observed, with values of the ratio well above those for large k. It seems like the value of ξ_t used has an impact in this ratio, so that the smaller value $\xi_t = 1/10$ causes larger ratio values. Surprisingly, for large values of k ($k \geq 10^6$), the ratios observed are almost exactly the constant factors of k obtained from the analysis [7]. (Recall that all the analyses we refer to are with high probability while the simulation results are averages.) This may indicate that the analysis with high probability is very tight and that the term $O(\log^2(1/\varepsilon))$ that appears in the complexity expression is mainly relevant for moderate values of k. The ratio obtained for large k by Log-Fails Adaptive with $\xi_t = 1/10$ is the smallest we have obtained in the set of simulations. Loglog-iterated Back-off, on its hand, seems to have a constant ratio of around 10. In reality this ratio is not constant but, since it is sublogarithmic, this fact can not be observed for the (relatively small) values of k simulated.

Regarding the ratios obtained for the algorithms proposed in this paper, they seem to show that the constants obtained in the analyses (with high probability) are very accurate. Starting at moderately large values of k (10^3 and up) the ratio for One-Fail Adaptive becomes very stable and equal to the value of 7.4 obtained in the analysis. The ratios for the Exp Back-on/Back-off simulations, on their hand, move between 4 and 8, while the analysis for the value of δ used yields a constant factor of 14.9. Hence, the ratios are off by only a small constant factor. To appreciate these values it is worth to note that the smallest ratio expected by any algorithm in which nodes use the same probability at any step is e, so these values are only a small factor away from this optimum ratio. In summary, the algorithms proposed here have small and stable ratios for all values of k considered.

6 Conclusions and Open Problems

In this work, we have shown optimal randomized protocols (up to constants) for static k-selection in Radio Networks that do not require any knowledge on the number of contenders. Future work includes the study of the dynamic version of the problem when messages arrive at different times under the same model, either assuming statistical or adversarial arrivals. The stability of monotonic strategies (exponential back-off) has been studied in [2]. In light of the improvements obtained for batched arrivals, the application of non-monotonic strategies to the dynamic problem is promising.

References

1. Balador, A., Movaghar, A., Jabbehdari, S.: History based contention window control in ieee 802.11 mac protocol in error prone channel. Journal of Computer Science 6, 205–209 (2010)
2. Bender, M.A., Farach-Colton, M., He, S., Kuszmaul, B.C., Leiserson, C.E.: Adversarial contention resolution for simple channels. In: 17th Ann. ACM Symp. on Parallel Algorithms and Architectures, pp. 325–332 (2005)
3. Capetanakis, J.: Tree algorithms for packet broadcast channels. IEEE Trans. Inf. Theory IT-25(5), 505–515 (1979)
4. Chlebus, B.S.: Randomized communication in radio networks. In: Pardalos, P.M., Rajasekaran, S., Reif, J.H., Rolim, J.D.P. (eds.) Handbook on Randomized Computing, vol. 1, pp. 401–456. Kluwer Academic Publishers, Dordrecht (2001)

5. Clementi, A., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: Proc. of the 12th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 709–718 (2001)
6. 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)
7. Fernández Anta, A., Mosteiro, M.A.: Contention resolution in multiple-access channels: k-selection in radio networks. Discrete Mathematics, Algorithms and Applications 2(4), 445–456 (2010)
8. Fernández Anta, A., Mosteiro, M.A., Muñoz, J.R.: Unbounded Contention Resolution in Multiple-Access Channels. arXiv:1107.0234v1 [cs.DC] (July 2011)
9. Gerèb-Graus, M., Tsantilas, T.: Efficient optical communication in parallel computers. In: 4th Ann. ACM Symp. on Parallel Algorithms and Architectures, pp. 41–48 (1992)
10. Greenberg, A., Winograd, S.: A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. Journal of the ACM 32, 589–596 (1985)
11. Greenberg, R.I., Leiserson, C.E.: Randomized routing on fat-trees. Advances in Computing Research 5, 345–374 (1989)
12. Gusella, R.: A measurement study of diskless workstation traffic on an ethernet. IEEE Transactions on Communications 38(9), 1557–1568 (1990)
13. Hayes, J.F.: An adaptive technique for local distribution. IEEE Trans. Comm. COM-26, 1178–1186 (1978)
14. Indyk, P.: Explicit constructions of selectors and related combinatorial structures, with applications. In: Proc. of the 13th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 697–704 (2002)
15. Komlòs, J., Greenberg, A.: An asymptotically nonadaptive algorithm for conflict resolution in multiple-access channels. IEEE Trans. Inf. Theory 31, 303–306 (1985)
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. 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)
18. Leland, W.E., Taqqu, M.S., Willinger, W., Wilson, D.V.: On the self-similar nature of ethernet traffic (extended version). IEEE/ACM Transactions on Networking 2, 1–15 (1994)
19. Martel, C.U.: Maximum finding on a multiple access broadcast network. Inf. Process. Lett. 52, 7–13 (1994)
20. Mikhailov, V., Tsybakov, B.S.: Free synchronous packet access in a broadcast channel with feedback. Problemy Peredachi Inform 14(4), 32–59 (1978)
21. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
22. Nakano, K., Olariu, S.: A survey on leader election protocols for radio networks. In: Proc. of the 6th Intl. Symp. on Parallel Architectures, Algorithms and Networks (2002)
23. Willard, D.E.: Log-logarithmic selection resolution protocols in a multiple access channel. SIAM Journal on Computing 15, 468–477 (1986)

Deterministic and Energy-Optimal Wireless Synchronization[⋆]

Leonid Barenboim[1,⋆⋆], Shlomi Dolev[1,⋆⋆⋆], and Rafail Ostrovsky[2,†]

[1] Department of Computer Science, Ben-Gurion University of the Negev, POB 653, Beer-Sheva 84105, Israel
{leonidba,dolev}@cs.bgu.ac.il

[2] Computer Science Department and Department of Mathematics, University of California Los Angeles, 90095, USA
rafail@cs.ucla.edu

Abstract. We consider the problem of clock synchronization in a wireless setting where processors must minimize the number of times their radios are used, in order to save energy. Energy efficiency is a central goal in wireless networks, especially if energy resources are severely limited, as occurs in sensor and ad-hoc networks, and in many other settings. The problem of clock synchronization is fundamental and intensively studied in the field of distributed algorithms. In the current setting, the problem is to synchronize clocks of m processors that wake up in arbitrary time points, such that the maximum difference between wake up times is bounded by a positive integer n. (Time intervals are appropriately discretized to allow communication of all processors that are awake in the same discrete time unit.) Currently, the best-known results for synchronization for single-hop networks of m processors is a randomized algorithm due to Bradonjic, Kohler and Ostrovsky [2] of $O\left(\sqrt{n/m} \cdot \textit{poly-log}(n)\right)$ radio-use times per processor, and a lower bound of $\Omega\left(\sqrt{n/m}\right)$. The main open question left in their work is to close the poly-log gap between the upper and the lower bound and to

⋆ A full version of this paper with full proofs and pseudocode of all procedures is available on-line [1].

⋆⋆ Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities. Additional funding was provided by the Lynne and William Frankel Center for Computer Sciences.

⋆⋆⋆ Research supported in part by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Microsoft, Deutsche Telekom, VeriSign .Com grant, Israeli Defense Secretary (MFAT), US Air-Force and Rita Altura Trust Chair in Computer Sciences.

† Supported in part by NSF grants 0830803, 09165174 and 1016540, US-Israel BSF grant 2008411, grants from OKAWA Foundation, IBM, Lockheed-Martin Corporation and the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

de-randomize their probabilistic construction and eliminate error probability. This is exactly what we do in this paper. That is, we show a *deterministic* algorithm with radio use of $\Theta\left(\sqrt{n/m}\right)$, which exactly matches the lower bound proven in [2], up to a small multiplicative constant. Therefore, our algorithm is *optimal* in terms of energy efficiency and completely resolves a long sequence of works in this area [2, 11–14]. Moreover, our algorithm is optimal in terms of *running time* as well. In order to achieve these results we devise a novel *adaptive* technique that determines the times when devices power their radios on and off. This technique may be of independent interest.

In addition, we prove several lower bounds on the energy efficiency of algorithms for *multi-hop networks*. Specifically, we show that any algorithm for multi-hop networks must have radio use of $\Omega(\sqrt{n})$ per processor. Our lower bounds holds even for specific kinds of networks such as networks modeled by unit disk graphs and highly connected graphs.

1 Introduction

Problem Description and Motivation: In wireless networks in general, and in sensor and ad hoc networks in particular, minimizing energy consumption is a central goal. It is often the case that energy resources are very limited for such networks. Consider, for instance, a sensor network whose processors are fed by solar energy. In such cases devising energy efficient algorithms becomes crucial. A significant energy use of a processor takes place when its radio device is on. Then, it is able to communicate with other processors in its transmission range whose radio devices are also turned on. However, it wastes significantly more energy than it would waste if its radio device was turned off. For example, in typical sensor networks [16] listening to messages consumes roughly as much energy as fully utilizing the CPU, and transmitting consumes up to 2.5 times more energy. Moreover, if a processor runs in an idle mode, and its radio device is off, it consumes up to 100 times less energy than it would consume if its radio device was on. Therefore, the time that a processor can operate using an allocated energy resource largely depends on how often its radio is turned on.

Processors in a wireless network may wake up at somewhat different time points. For example, in the sensor network powered by solar energy, processors wake up in the morning when there is enough light projected on their solar cells. If the processors are spread over a broad area, then there is a difference in the wake up times. The processors' clocks start counting from zero upon wake up. Since there is a difference in wake up times, the clocks get out of synchronization. However, many network tasks require that all processors agree on a common time counting. In such tasks processors are required to communicate only in certain time points, and may be idle most of the time. If the clocks are not synchronized, a certain procedure has to be invoked by each processor in order to check the status of other processors. During this procedure processors may turn their radio on constantly, resulting in a major waste of energy. Therefore, clocks must be synchronized upon wake up in order to save energy and to allow the execution of timely mannered tasks. The clock synchronization itself must be as efficient as possible in terms of energy use. It is desirable that among all possible strategies, each processor selects the strategy that minimizes its radio use. The *energy efficiency* of a processor is the number of time units in which its radio device is turned on.

In this paper we devise energy efficient clock synchronization algorithms. The goal of a clock synchronization algorithm is setting the logical clocks of all processors such that all processors show the same value at the same time. In order to achieve this goal, each processor executes an adaptive algorithm, which determines the time points (with respect to its local clock) in which the processor will turn its radio device on, for a fixed period of time. Once a processor's radio device is on, it is able to communicate with other processors in its range whose radio devices are also on at the same time interval [1]. Based on the received information a processor can adjust its logical clock, and determine additional time intervals in which its radio device will be turned on. This process is repeated until all processors are synchronized.

Our Results: We consider single-hop networks of m processors, such that the maximum difference between processors wake up times is n. (Henceforth, *the uncertainty parameter*.) We devise several *deterministic* synchronization algorithms, the best of which has radio efficiency $O(\sqrt{n/m})$ per processor. Our results improve the previous state-of-the-art algorithms devised by Bradonjic et al. [2]. In [2] *randomized* algorithms for synchronization single-hop networks were devised, whose energy efficiency is $O(\sqrt{n/m} \cdot polylog(n))$ per processor. Therefore, our *deterministic* results improve the best previous *randomized* results by a polylogarithmic factor. Moreover, Bradonjic et al. proved lower bounds of $\Omega(\sqrt{n/m})$ per processor for the energy efficiency of any deterministic clock synchronization algorithm for single-hop networks. Hence, our algorithms are optimal in terms of radio use up to constant factors.

We close the gap between the performance of the currently best-known randomized and deterministic algorithms for this problem. This is particularly interesting, because in many cases there exist (possibly inherent) significant gaps between randomized and deterministic algorithms. Notable examples are consensus where there is no deterministic solution, but there is a randomized one, or Maximal Independent Set and $O(\Delta)$-coloring for which the gaps between best-known randomized and deterministic algorithms are exponential. In addition, our algorithms do not employ heavy machinery, as opposed to [2], where expanders and sophisticated probabilistic analysis are employed. In contrast, we devise a combinatorial construction that quickly "spreads" processors' radio use approximately equally in time, which surprisingly allows them to synchronize more efficiently via chaining synchronization messages with each other. As a

[1] In our model more than one processor can transmit at a time without interference. This can be achieved using standard multiple access schemes (CDMA, FDMA). Alternatively, one can use traditional radio broadcast *over* our scheme. (See, e.g., [2].) The problems of channel multiplexing, and of radio broadcast, are different from the problem we address in this paper.

result, our algorithm is also optimal in terms of *running time* (up to a small constant factor). It runs in $O(n)$ time, and improves the running time of [2], which is $O(n \cdot polylog(m))$.

We also prove lower bounds for *multi-hop* networks. We show that any deterministic synchronization algorithm for an m-processor multi-hop network must have total radio use $\Omega(m \cdot \sqrt{n})$. In [2] a simple deterministic algorithm for 2-processor network was devised with energy efficiency $O(\sqrt{n})$ per processor. It is extendable to m-vertex networks, in the sense that each processor learns the differences between its clock and the clocks of its neighbors. The total radio efficiency of the extended algorithm is $O(m \cdot \sqrt{n})$. As evident from our results it is far from optimal for single-hop networks. However, for multi-hop networks, its total radio efficiency $O(m \cdot \sqrt{n})$ is the best possible up to constant factors. Our lower bounds hold even for very specific network types such as unit disk graph and highly connected graphs.

High-Level Ideas: In the synchronization algorithm for m processors devised in [2] each processor determines by itself the time points in which it turns its radio on. The decision of a processor does not depend by any means on the decisions of other processors. Such a non-adaptive strategy makes the algorithm sub-optimal unless the number of processors is constant. Moreover, the decisions are made using randomization, and, consequently, the algorithm may fail. (However, the probability of failure is very low, since it is exponentially in n close to zero.) In contrast, our algorithms are *deterministic* and *adaptive*. In our algorithms, periodically, each processor deterministically decides for the time points in the future in which it will turn its radio on. Each decision is made based on all the information the processor has learnt from communicating with other processors before the time of decision. Such a strategy decreases the number of redundant radio uses. In other words, the radio of a processor is used only if this processor is essential for synchronization, and no other processor can be used instead. Since all processors use this strategy, the radio use of each processor is as small as possible.

Let V be an m-vertex set representing the processors of the network, and E an initially empty edge set. Each time a pair of processors $u, v \in V$ communicate with each other, add the edge (u, v) to E. Once the graph $\mathcal{G} = (V, E)$ becomes connected, all m processors can be synchronized. Each time a processor turns its radio on, it communicates with other processors that also turn their radio on in the same time. Consequently, additional edges are added to E, and the graph $\mathcal{G}$ changes. In all time points the graph $\mathcal{G}$ consists of clusters. Initially, each vertex is a cluster, and clusters are merged as time passes. Each time a new cluster is formed, the clocks of the processors in the cluster are synchronized using our cluster-synchronization procedures. Next, each processor selects exclusive (with respect to other processors in the cluster) time points in the future in which its radio will be turned on. For a sufficient number of points, such a selection guarantees that one of the processors in the cluster will turn the radio on in the same time with another processor from another cluster. This results in merging of the clusters. Our algorithms cause all clusters to merge into a single unified cluster that contains all m vertices very quickly.

Related Work: Clock synchronization is one of the most intensively studied and fundamentally important fields in distributed algorithms. [3, 4, 6–8, 10–15, 17, 18]. The aspect of energy efficiency of clock synchronization algorithms was concerned in most of these works. In [14] Polastre et al. devised an algorithm with energy efficiency $O(n)$ per processors. Each processor simply turns its radio on for $n + 1$ consecutive time units upon wake up. Since the maximum difference between wake up points is n, this guarantees that all processors are synchronized. More efficient solutions were devised by McGlynn and Borbash [12], Palchaudhuri and Johnson [13], and by Moscibroda, Von Rickenbach, and Wattenhofer [11]. In these solutions, each processor turns its radio on for $O(\sqrt{n})$ time units that are randomly selected. Their correctness is based on the birthday paradox, according to which there exists a time point that is selected by two processor with high probability. In this time point both processors turn their radio on and are able to synchronize.

2 Preliminaries

The Setting: We use the following abstract model of a wireless network. We remark that although this model is quite strong, it is sufficiently expressive to capture a more general case as shown in [1, 2]. Global time is expressed as a positive integer, and available for analysis purposes only. The network is modeled by an undirected m vertex graph $G = (V, E)$. The processors of the network are represented by vertices in V, and enumerated by $1, 2, ..., m$. For each pair of processors u and v residing in the communication range of each other there is an edge $(u, v) \in E$. Communication is performed in discrete rounds. Specifically, time is partitioned into units of equal size, such that one time unit is sufficient for a transmitted message to arrive at its destination. (At the physical level this can be relaxed such that communication is possible if two processors turn their radio on during intervals that overlap for at least one time unit.) A processor wakes up in the beginning of a time unit, and its physical and logical clocks start counting from zero. The clocks of all processors tick with the same speed, and are incremented in the beginning of each new time unit. The wake up time of the processors, and, consequently, the clock values in a certain moment may differ. However, the maximum difference between the wake up times of any two processors is bounded by an integer n, which is known to all processors. (In other words, each processor wakes up with an integer shift in the range $\{0, 1, ...n\}$ from global time 0.) See [1] for a discussion on more general cases. Specifically, the wake up shifts may be non-integers, and the clock speeds may somewhat differ, as long as the ratio of different speeds is bounded by a constant.

Each processor has a radio device that is either on or off during each time unit. If the radio device is off, its energy consumption is negligible. The energy efficiency of an algorithm is the number of time units during which the radio device of a processor is on. A pair of processors $(u, v) \in E$ are able to communicate in a certain time unit t (with respect to global time) only if the radio devices of both u and v are turned on during this time unit.

Algorithm Representation: The running time $f(n, m)$ of an algorithm is the worst case number of time units that pass from wake up until the algorithm terminates. The algorithm specifies initial fixed time points for a processor to turn its radio on. In addition, it adaptively determines new time points each time a processor turns its radio on. The time points are determined by assigning strings to processors as follows. The strings of the m processors are represented using a two dimensional array A. The array A contains m rows. For $i = 1, 2, ..., m$, The ith row belongs to the ith processor. The number of columns of A is $n + f(n, m)$. All cells of A are set to 0, except the cells that are explicitly set to 1. (Initially, all cells are set to 0.) The algorithm specifies an initial fixed string S_i for each processor i. For $i = 1, 2, ..., m$, suppose that processor i wakes up at time t_i, $0 \leq t_i \leq n$, with respect to global time. Then the ith row of A is initialized as follows. For $j = 0, 1, ..., |S_i| - 1$, set $A[i][t_i + j] = S_i[j]$.

For $j = 0, 1, 2...$, at local time j, a processor i accesses the cell $A[i][t_i + j]$. A processor i turns its radio device on at local time $k \geq 0$ if and only if $A[i][t_i + k] = 1$. If at global time t the radio device of a processor i is on, then it can communicate with all processors j in its communication range for which $A[j][t] = 1$. Based on the received information, processor i deterministically decides whether to update cells in the row $A[i]$. It can update, however, only cells that represent time points in the future, i.e., cells $A[i][t']$, for $t' > t$. Observe, however, that processor i is unaware both of global time and the shift t_i. (In particular, it is unaware of the index of the cell it is accessing in the row $A[i]$.) The algorithm terminates once all processors detect a column of ones, i.e., a column ℓ such that for all $1 \leq j \leq m$, it holds that $A[j][\ell] = 1$. (Once all processors detect a column of ones, they all turn their radio on in the same time, and synchronize their clocks.) A clock synchronization algorithm $\mathcal{A}$ is correct if for all $i = 1, 2, ..., m$, for all shifts t_i, $t_i \in \{0, 1, 2, ..., n\}$, once $\mathcal{A}$ is executed by all processors there exists a column ℓ such that for all $j = 1, 2, .., m$, $A[j][\ell] = 1$.

Building Blocks and Definitions: A *radio use policy* is a protocol for a processor $i \in \{1, 2, ..., m\}$ that determines the local time points in which the processor i turns its radio on. For $r = 0, 1, 2, ...$, in the beginning of time unit r from wake up, the processor i decides whether to turn its radio device on as explained above[1].

For a fixed string s over the alphabet $\{0, 1\}$ and a positive integer t, an (s, t)-*radio-use policy* of a processor i determines the local time units in which i turns its radio on. For a processor i that wakes up at global time t_i, we say that processor i *performs an (s,t)-radio use policy* if it sets $A[i][t_i + t + j] = s[j]$, for $j = 0, 1, ..., |s| - 1$, and turns its radio device on accordingly. (Recall that processor i turns its radio device on at local tick k if and only if $A[i][t_i + k] = 1$.) The processor starts performing the policy at global time $t_i + t$. It completes the

[1] The decision process can also be performed using a decision tree.

policy at global time $t_i + t + |s| - 1$. During this period a processor may select new time points in the future in which additional policies will be performed.

Next, we define the notion of *length*, *covering-weight*, and *covering-density* of a policy. These definitions are used in the correctness analysis of the algorithms. The *length* of an (s,t)-radio-use policy p, denoted $len(p)$, is the difference between the positions of the first and last '1' in s plus one. (In other words, if j is the smallest index such that $s[j] = 1$, and k is the largest index such that $s[k] = 1$, then $len(p) = k - j + 1$.) Intuitively, the length of a policy is the time duration required for performing the policy. For the (s,t)-radio-use policy p, the string s is a concatenation of two substring $s' \circ s''$, defined by p. The substring s' is called the *initial part* of s, and the substring s'' is called the *main part* of s. We say that i *performs the initial part of* p in global time t' if it performs the policy p, and the global time t' satisfies $t + t_i \leq t' \leq t + t_i + |s'| - 1$. If i performs the policy p, and the global time t' satisfies $t + t_i + |s'| - 1 < t'$, we say that i *performs the main part of* p.

We say that two processors i and j *overlap* if there is a global time point t' in which both processors turn their radio on. Two processors u and v can communicate (either directly or indirectly) if they overlap, or if there exist a series of processors $w_1, w_2, ..., w_k$, $w_1 = u, w_k = v$ such that w_i overlaps with w_{i+1}, for $i = 1, 2, ..., k-1$. If such a series does not exist, we say that there is a point of discontinuity between u and w. A *point of discontinuity* is a global time point t' in which either (1) there is no processor that performs a radio use policy, or (2) each processor that do perform a radio use policy, completes it in time t'. A global time interval (s', t') is *continuous* if there are no points of discontinuity in it. For a continuous interval (s', t') such that s' and t' are discontinuity points, all processors performing a radio use policy during the interval (s', t') form a *cluster* c. In this case we say that c *covers* the interval (s', t'). The *length* of a cluster c that covers an interval (s', t'), denoted $len(c)$, is $t' - s' + 1$.

Each processor in a cluster adds weight to the cluster. Consequently, clusters containing many processors are heavier than clusters containing few processors. The *covering-weight* of a cluster c, denoted $cwet(c)$, is the sum of lengths of policies of processors contained in c. Consider two clusters c and c' with the same covering-weight, but such that the length of c is much shorter then the length of c'. Therefore, c' covers a much longer time interval. We show later in this paper that clusters that cover longer intervals are 'better' in a certain way. Consequently, c' is better than c, although they have the same covering weight. On the other hand, a short and light cluster may be better than a long and heavy one. Therefore, neither the length nor the covering-weight of a cluster are expressive enough to determine how 'good' a cluster is. Hence, we add the notion of covering-density, which is the ratio between covering-weight and length of a cluster. The *covering-density* of a cluster c, denoted $cden(c)$, is $\frac{cwet(c)}{len(c)}$. Clusters of lower covering-density are considered as better clusters. (Observe that these definitions are different from the usual definitions of string weight and density in which only the number of ones in the string are counted.)

Next, we give similar definitions for intervals. The *length* of an interval $q = (s', t')$, denoted $len(q)$, is $t' - s' + 1$. Suppose that during interval q there are ℓ policies that are performed. (Possibly, some have started before time s', and some have ended after time t'.) Let $q_1, q_2, ..., q_\ell$ be the intervals contained in q in which the main parts of the policies are performed. The *covering-weight* of an interval q, denoted $cwet(q)$, is $\Sigma_{i=1}^{\ell} len(q_i)$. The *covering-density* of the interval, denoted $cden(q)$, is $\frac{cwet(q)}{len(q)}$.

3 Synchronization Algorithms for Single-Hop Networks

Procedure Synchronize
In this section we present a *deterministic* synchronization algorithm for complete graphs on m vertices with energy efficiency $O((\sqrt{n/m}) \log n)$ per processor. In the next section we devise an algorithm with energy efficiency $O(\sqrt{n/m})$ per processor. As a first step, we define the following basic radio use policy for a processor, consisting of two parts. Starting from local time t, for a given integer $k > 0$, turn the radio devise on for k consecutive time units. (Henceforth, *initial part.*) Then, for the following k^2 time units, turn the radio on only once in each k consecutive time units. (Henceforth, *main part.*) In other words, starting from the beginning of the main part, the radio is turned on during time units $k, 2k, 3k, ..., k^2$. This completes the description of the policy. It henceforth will be referred as *k-basic policy.* The string s of the policy is defined by $s[i] = 1, s[(i+2) \cdot k - 1] = 1$ for $0 \leq i < k$. The length of a k-basic policy is $k + k^2$, but the number of time units in which the radio is used is only $2k$. We remark that the k-basic policy is defined for any integer $k > 0$, but we employ policies in which $k = \Theta(\sqrt{n/m})$.

Consider a pair of processors u and v that wake up at the beginning of global time units t_u and t_v, respectively, such that $t_u < t_v$. Suppose that both processors use the k-basic policy p upon wake up, and that $t_v - t_u < len(p)$. Then, there is a global time unit t in which both processors turn their radio devices on. In this case we say that the processors *overlap.* We summarize this fact in the next lemma.

Lemma 3.1. *Suppose that processors u and v wake up at global time points $t_u < t_v$, such that $t_v - t_u < len(p)$, and execute the k-basic policy p upon wake up, for an integer $k > 0$. Then u and v overlap.*

Any two overlapping processors synchronize their clocks as follows. Each processor executes the following procedure called *Procedure Early-Synch.* During its execution the processor that began performing its radio policy later among the two is synchronized with the other processor. In other words, the later processor updates its logical clock to be equal to the logical clock of the earlier processor. (Observe that the clock value of the later processor is not greater than that of the earlier processor, therefore, clocks do not go backwards.) To this end, each processor maintains the local variables Id, τ, J, where Id is the unique identity number of the processor, τ is the local clock value, and J is the number of time units passed since the processor began performing the current radio policy. The variable τ is updated in each time unit by reading the logical clock value and assigning it to τ. The variable J is set to 0 each time the processor starts a radio use policy, and is incremented in each time unit. Each time a processor turns its radio on, it transmits the message (Id, τ, J). Once a processor u receives a message (Id_v, τ_v, J_v) form a processor v, processor u determines whether it began its radio policy after v did. If so, u updates its local clock to τ_v, and its variable J_u to J_v. If both processors began their policy at the same time, then the clocks are synchronized to the clock of the processor with the greater Id. The procedure returns the value $J = J_u = J_v$. This completes its description.

Lemma 3.2. *Procedure Early-Synch executed by two overlapping processors synchronize their clocks.*

Procedure Early-Synch can be generalized for synchronizing a cluster containing an arbitrary number of processors. Recall that all processors in the cluster perform their policies in a time interval (s', t') containing no discontinuity points. Hence, a message from a processor u can be delivered to all processors that begin performing their policy after u does so. The message is received directly by all processors that overlap with u, and is propagated in a rely-race manner to other processors. In this way all the processors in the cluster can be synchronized with the processor that was the first to start performing its policy.

The generalized procedure is called *Procedure Cluster-Synch.* During its execution all processors $u \in V$ perform the k-basic policy. A vertex u starts performing its policy at local time point T_u that is passed to the procedure as an argument. (The argument is passed by another procedure that invokes Procedure Cluster-Synch, which is described later in this section.) Each processor u initializes a counter J_u that is set to 0 once the policy starts, and is incremented by 1 in each time unit. Recall that the local clock of u is represented by the variable τ_u. Each time a radio device of a processor u is on it transmits the message (Id_u, τ_u, J_u). For each received message (Id_v, τ_v, J_v) from a vertex v, if $(J_u < J_v)$ or $(J_u = J_v$ and $Id_u < Id_v)$, then u updates its clock to τ_v and its counter J_u to J_v. This completes the description of Procedure Cluster-Synch. Its pseudocode is provided below. Its properties are summarized in Lemma 3.3. Its correctness follows from the observation that all processors eventually synchronize their counters J with the counter of the earliest processor in the cluster.

Algorithm 1. Procedure Cluster-Synch(T_u,k)

An algorithm for processor u.

```
1: Perform the $k$-basic policy starting from local time $T_u$
2: $J$ := Early-Synch()
3: return $J$
```

Lemma 3.3. *For a fixed $k > 0$, suppose that processors $v_1, v_2, ..., v_\ell$ perform Procedure Cluster-Synch(T_{v_i}, k), with the parameters $T_{v_1}, T_{v_2}, ..., T_{v_\ell}$, respectively. If in the resulting execution the processors $v_1, v_2, ..., v_\ell$ form a cluster, then $v_1, v_2, ..., v_\ell$ synchronize their clocks to the clock of the earliest processor v_1.*

Next, we consider the most general problem in which m processors wake up at arbitrary global time points in the time interval $[0, n]$. If each processor performs the k-basic policy upon wake up, then several clusters may be produced. The processors in each cluster can be synchronized using procedure Cluster-Synch. However, the execution of procedure Cluster-Synch will not synchronize processors from distinct clusters since any two distinct clusters are separated by a discontinuity point. We devise a procedure, called *Procedure Synchronize* that merges these clusters gradually, until only a single cluster remains. To this end, the parameter k is selected to be large enough to guarantee that certain clusters have large covering-density. The processors in a cluster with large covering-density schedule the next policy execution times in a specific way that enlarges the length of the cluster to the maximal extent. Somewhat informally, the cluster is extended roughly equally to both of its sides. In other words, there is an integer $L > 0$ such that in the next phase the cluster begins L time units earlier than in the previous phase, and terminates L units later than in the previous phase. For a precise definition see Algorithm 2, line 14. The extension of the cluster to both of its sides prevents time drifts, and, consequently, in each phase some clusters overlap. Overlapping clusters are merged into fewer clusters of greater covering-weight.

Algorithm 2. Procedure Flatten(J_u, k)

A protocol for a vertex u, executed once u completes the initial part of its policy.

```
1: /*** First stage ***/
2: $J := J_u$
3: wait for $2n - J$ time units
4: /*** Second stage ***/
5: $B := \{(Id_u, J)\}$
6: transmit $(Id_u, J)$
7: for each received message $m = (Id_v, J')$ do
8:     $B := B \cup \{m\}$
9: end for
10: $B' :=$ sort $B$ by Ids in ascending order
11: $len(c) := (\max\{J' | (Id, J') \in B'\})$
12: $\mu :=$ the position of $(Id_u, J)$ in $B'$
13: $\ell := |B|$
14: $next := \left\lfloor 2n + \tau_u + \frac{len(c) - \ell \cdot k^2}{2} + \mu \cdot k^2 \right\rfloor$
15: return $next$   /* returned locally to the caller of this procedure */
```

The procedure for extending the length of a cluster is called *Procedure Flatten*. It is executed by processors in a synchronized cluster c, and proceeds in two stages. The first stage (See Algorithm 2, lines 1-3) is executed by each processor u in the cluster once the processor u is synchronized with the first processor of the cluster v_1. (In other words, once u completes the initial part of its k-basic policy.) Then the counters J of u and v_1 are also synchronized. A processor u

schedules the next execution of its policy to be executed in $2n - J$ time units. The second stage (Algorithm 2, lines 4-15) is executed once u performs the policy the next time. Observe that it is executed in the same time by all processors in the cluster. All processors of the cluster turn their radio on and learn the number of processors in the cluster, their Ids, and the length of the cluster $len(c)$ with respect to the first stage. (We describe how to determine $len(c)$ shortly.) Each processor sorts the Ids and finds its position μ in the sorting. If the current local time is τ, and the number of processors in the cluster is ℓ, it schedules the next policy execution to local time $\left\lfloor 2n + \tau + \frac{len(c) - \ell \cdot k^2}{2} + \mu \cdot k^2 \right\rfloor$, and returns this value.

The length of the cluster $len(c)$ is equal to the difference between the global time points of the beginning and the end of the cluster. Thus, the length $len(c)$ is determined by the latest processor in c. Once the latest processor v_ℓ completes its policy in the first stage, its counter J_ℓ (which is synchronized with the counter of the earliest processor) is equal to the number of time units passed since the cluster has started. Once v_ℓ completes its policy, the entire cluster c is completed. Hence, at that moment, it holds that $len(c) = J_\ell$. All processors learn this value in the second stage. (See step 11 in the pseudocode of Algorithm 2.) This completes the description of the procedure. Its properties are summarized below.

Lemma 3.4. *Suppose that Procedure Flatten is executed by a cluster c of ℓ processors that is formed in a global time interval $[p, q]$. Then*
(1) The second stage of Procedure Flatten is performed at global time $p + 2n$ by all processors of c.
(2) Performing the policies by the scheduling of the second stage forms a cluster c' of length $\ell \cdot k^2$.
(3) The cluster c' covers an interval that contains the interval $[4n + \frac{p+q}{2} - \frac{\ell \cdot k^2}{2}, 4n + \frac{p+q}{2} + \frac{\ell \cdot k^2}{2}]$.

In order to synchronize m processors that wake up at arbitrary times from the interval $[0, n]$, set $k = \left\lceil \sqrt{8 \cdot n/m} \right\rceil$. Procedure Synchronize is performed in phases as follows. For $i = 1, 2...$, the ith phase starts in global time $(i - 1) \cdot 4n$. In each phase, two iterations are performed. Initially, in the first iteration of the first phase, each processor performs the k-basic radio policy upon wake up. Consequently, clusters are formed in the interval $[0, 2n]$. Each cluster is synchronized using Procedure Cluster-Synch. In the second iteration of the first phase, Procedure Flatten is performed. Then the next phase starts. In the first iteration of each phase, the k-basic policy is performed by each processor starting from a time point that was scheduled for it in the previous phase by Procedure Flatten. Consequently, new clusters are formed and synchronized. In the second iteration, Procedure Flatten is performed, and schedulings for the next phase are determined. Procedure Synchronize terminates once the interval $[i \cdot 4n, i \cdot 4n + 2n]$ is continuous, for an integer $i > 0$. A continuous cluster of length at least $2n$ necessarily contains all m processors. Finally, Procedure Cluster-Synch is executed causing all m processors to synchronize. This completes the description of Procedure Synchronize. The pseudocode is provided below.

Algorithm 3. Procedure Synchronize()

```
An algorithm for a processor $v$
1: $k = \left\lceil \sqrt{8 \cdot n/m} \right\rceil$ ; $\tau = 0$
2: for $i = 1, 2, ..., \lceil \log n \rceil$ do
3:    $J$ := Cluster-Synch($\tau$, $k$)
4:    $\tau$ := Flatten($J, k$)
5: end for
6: Cluster-Synch($\tau$, $k$)
```

Procedure Synchronize preserves cluster distances in each phase in the following sense. Suppose that processors u and v wake up at global times t_u and t_v respectively. Then, for $i = 1, 2, ..., \log n$, there are clusters c_i and c'_i such that c_i covers an interval containing the point $(t_u + 4n \cdot i)$, and c'_i covers an interval containing the point $(t_v + 4n \cdot i)$. Moreover, the cluster c_i contains the processor u, and the cluster c'_i contains the processor v. This observation, which is a consequence of Lemma 3.4, is summarized below.

Corollary 3.5. *Suppose that a processor v performs the k-basic policy in time $t \in [0, 2n]$. If a cluster c covers an interval containing the time point $(t + 4n \cdot i)$ for some integer $i > 0$, then c contains v.*

In each phase of Procedure Synchronize, after the execution of Procedure Flatten, the sum of lengths of produced clusters is at least $k^2 \cdot m > 2n$. Consequently, at least two clusters overlap in each phase, and the number of clusters is decreased in each phase. Hence, it is obvious that m phases are sufficient to merge all clusters into a single cluster. However, the merging process is actually much faster. The next Lemma states that after $\log n$ phases there is a single cluster containing all m processors.

Lemma 3.6. *Once Procedure Synchronize is executed, the global time interval $[\lceil \log n \rceil \cdot 4n, \lceil \log n \rceil \cdot 4n + 2n]$ is continuous.*

By Corollary 3.5 and Lemma 3.6, all m processors are synchronized during global time interval $[\lceil \log n \rceil \cdot 4n, \lceil \log n \rceil \cdot 4n + 2n]$. Each processor performs the k-basic policy a constant number of times in each phase. Hence, in each phase, the number of time units in which each processor turns its radio on is $O(k) = O(\sqrt{n/m})$. The properties of Procedure Synchronize are summarized below.

Theorem 3.7. *Procedure Synchronize performs clock synchronization of m processors waking up at arbitrary time points from the interval $[0, n]$. The energy efficiency of each processor is $O(\sqrt{n/m} \cdot \log n)$. The running time of Procedure Synchronize is $O(n \log n)$.*

Procedure Dynamic-Synch

In this section we show that by using a more sophisticated procedures one can achieve energy efficiency of $O(\sqrt{n/m})$ per processor. We devise a procedure called *Dynamic Flattening.* The use of dynamic flattening allows completing the synchronization in two phases instead of $O(\log n)$ phases that are required by Procedure Synchronize that was devised in the previous section. The main difference of Procedure Dynamic Flattening comparing to Procedure Flatten is that the scheduling stage is performed during the first execution of the policy rather than in the end of a phase. This scheduling is performed only once, shortly after a processor wakes up. A processor schedules the next execution of its policy to the first available free interval, i.e., an interval in which no other processor is scheduled. To this end, a queue of processors is maintained by each cluster. Consequently, the next policy execution of each processor v is scheduled in such a way that the main part of v's policy does not overlap with any of the other $m-1$ processors when they execute the main parts of their policies after scheduling. (In contrast, in Procedure Flatten the new scheduling of phase i guarantees only that the main part of the policy of v does not overlap with any processor in the cluster containing v in phase i.) At time $2n$ at least $\frac{1}{2}m$ processors are scheduled one after the other to perform their policy. As a result, the global interval $[2n, 4n]$ is continuous. To guarantee that all m processor perform their policy during this interval, each processor performs an additional independent invocation of its policy at time $2n + 1$ from wake up.

The algorithm that employs this idea is called *Procedure Dynamic-Synch.*

Informal description of Procedure Dynamic-Synch (for each $v \in V$)
step 1. The vertex v sets $k := \left\lceil \sqrt{8 \cdot n/m} \right\rceil$, and performs the initial part of the k-basic policy.
step 2. If during step 1 one of the following holds: (i) v does not discover any other processor whose radio is turned on, or (ii) all discovered processors have waken up after v did, or have waken up at the same time as v but have smaller Ids than that of v,
then v initializes a cluster c and an empty queue q_c. The processor v enqueues itself on q_c and starts the main part of its policy once the initial part is complete.
step 3 (Dynamic Flattening). Otherwise, a queue q is already initialized and maintained by the processor u currently executing the main part of its policy. (The queue q was created by the earliest processor in the cluster and passed in a rely-race manner. We stress that u is not necessarily the earliest processor in the cluster.) Then v enqueues itself on q by communicating with u, and receives the number ℓ of processors that appear in q before v. Suppose that u has performed the main part of its policy for r rounds once communicating with v. Then v schedules the next k-basic policy execution such that the main part of its policy is executed in $(\ell-1)\cdot k^2-r$ time units. This guarantees that policies of processors in q are executed one after the other immediately, in the order they appear in q.
step 4. Once v completes executing its main part, it dequeues itself from q and passes q to the next scheduled processor (with which it necessarily overlaps).
step 5. Execute the k-basic policy at time $2n+1$ from wake up. (Independently of steps 1-4.) This completes the description of the procedure. Its properties are summarized below.

Lemma 3.8. *Suppose that m processors wake up during the global time interval $[0, n]$, and execute procedure Dynamic-Synch. Then it holds that:* **(1)** *for any pair of processors u and v, the time intervals in which their main parts are executed are distinct (i.e., have no common time points) in the global interval $[0, 2n]$,* **(2)** *each cluster c that covers an interval in $[0, 2n]$ satisfies that $cden(c) \leq 1$,* **(3)** *there exists a cluster c' that covers an interval containing the global time point $2n$. At global time $2n$ the queue of c' contains at least $m/2$ processors.*

By Lemma 3.8 (3), at global time point $2n$ at least $m/2$ processors are scheduled consequently. Hence the global interval $[2n, 4n]$ is continuous. All m processors execute their policy during this interval. Hence all m processors synchronize their clocks. Each processor execute the k-basic policy (fully or partly) 3 times.

Theorem 3.9. *Proc. Dynamic-Synch synchronizes the clock of m processors that wake up in the interval $[0, n]$. The energy efficiency per processor is $O(\sqrt{n/m})$.*

Each processor completes the execution of Procedure Dynamic-Synch within at most $4n$ time units from wake up. Therefore, the running time of the procedure is $O(n)$. Since in the worst case a processor may wait $\Omega(n)$ time units in order to exchange messages with any other processor, this running time is tight.

Theorem 3.10. *The running time of Proc. Dynamic-Synch is $O(n)$. It is optimal up to constant factors.*

We conclude this section by stating our lower bounds for *multi-hop* networks.

Theorem 3.11. *In any deterministic clock synchronization algorithm $\mathcal{A}$ for general graphs the sum of processors radio use is $\Omega(m \cdot \sqrt{n})$. Therefore, the average (and the worst case) energy efficiency of $\mathcal{A}$ per processor is $\Omega(\sqrt{n})$.*

Theorem 3.11 applies also to *unit disk graphs*. In the full version of this paper [1] we prove additional lower bounds, for an even narrower family of *ℓ-connected graphs*. Specifically, for this family of graphs, we prove that the energy efficiency of any deterministic algorithm is $\Omega(\sqrt{n/\ell})$.

4 Conclusion

In this paper we have devised optimal radio-use deterministic algorithms for clock synchronization in single-hop networks with energy efficiency $\Theta(\sqrt{n/m})$. We also proved lower bounds of $\Omega(\sqrt{n})$ for multi-hop networks. Our results suggest that in order to beat this bound of $\Omega(\sqrt{n})$, each neighborhood in the graph must be highly connected, containing no isolated regions. For wireless networks, this requires a certain level of uniformity in the processors distribution.

In [2] a deterministic synchronization algorithm was devised for two-processor networks with efficiency $O(\sqrt{n})$. This algorithm can be used also in multi-hop network in order to synchronize each processor with its neighbors. The energy efficiency in this case is $O(\sqrt{n})$ per processors. Somewhat surprisingly, our lower bounds imply that this simple approach is optimal in general multi-hop networks.

References

1. Barenboim, L., Dolev, S., Ostrovsky, R.: Deterministic and Energy-Optimal Wireless Synchronization (2010), http://arxiv.org/abs/1010.1112
2. Bradonjic, M., Kohler, E., Ostrovsky, R.: Near-Optimal Radio Use For Wireless Network Synchronization. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 15–28. Springer, Heidelberg (2009)
3. Boulis, A., Srivastava, M.: Node-Level Energy Management for Sensor Networks in the Presence of Multiple Applications. Wireless Networks 10(6), 737–746 (2004)
4. Boulis, A., Ganeriwal, S., Srivastava, M.: Aggregation in sensor networks: an energy-accuracy trade-off. Ad Hoc Networks 1(2-3), 317–331 (2003)
5. Bush, S.F.: Low-energy sensor network time synchronization as an emergent property. In: Proc. of the 14th International Conference on Communications and Networks, pp. 93–98 (2005)
6. Elson, J., Römer, K.: Wireless sensor networks: a new regime for time synchronization SIGCOMM Comput. Commun. Rev. 33(1), 149–154 (2003)
7. Fan, R., Chakraborty, I., Lynch, N.: Clock Synchronization for Wireless Networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 400–414. Springer, Heidelberg (2005)
8. Herman, T., Pemmaraju, S., Pilard, L., Mjelde, M.: Temporal partition in sensor networks. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 325–339. Springer, Heidelberg (2007)
9. Honda, N., Nishitani, Y.: The Firing Squad Synchronization Problem for Graphs. Theoretical Computer Sciences 14(1), 39–61 (1981)
10. Mills, D.L.: Internet time synchronization: the network time protocol. IEEE Transactions on Communications 39(10), 1482–1493 (1991)
11. Moscibroda, T., Von Rickenbach, P., Wattenhofer, R.: Analyzing the Energy-Latency Trade-Off During the Deployment of Sensor Networks. In: Proc. of the 25th IEEE International Conference on Computer Communications, pp. 1–13 (2006)
12. McGlynn, M., Borbash, S.: Birthday protocols for low energy deployment and flexible neighbor discovery in ad hoc wireless networks. In: Proc. of the 2nd ACM International Symposium on Mobile Ad Hoc Networking and Computing, pp. 137–145 (2001)
13. Palchaudhuri, S., Johnson, D.: Birthday paradox for energy conservation in sensor networks. In: Proc. of the 5th Symposium of Operating Systems Design and Implementation (2002)
14. Polastre, J., Hill, J., Culler, D.: Versatile low power media access for wireless sensor networks. In: Proc. of the 2nd International Conference on Embedded Networked Sensor Systems, pp. 95–107 (2004)
15. Schurgers, C., Raghunathan, V., Srivastava, M.: Power management for energy-aware communication systems. ACM Trans. Embedded Comput. Syst. 2(3), 431–447 (2003)
16. Shnayder, V., Hempstead, M., Chen, B., Allen, G., Welsh, M.: Simulating the power consumption of large-scale sensor network applications. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, pp. 188–200 (2004)
17. Sichitiu, M.L., Veerarittiphan, C.: Simple, accurate time synchronization for wireless sensor networks. In: IEEE Wireless Communications and Networking, WCNC 2003, vol. 2, pp. 1266–1273 (March 16-20, 2003)
18. Sundararaman, B., Buy, U., Kshemkalyani, A.D.: Clock synchronization for wireless sensor networks: a survey. Ad-hoc Networks 3(3), 281–323 (2005)

Leveraging Channel Diversity to Gain Efficiency and Robustness for Wireless Broadcast

Shlomi Dolev[1], Seth Gilbert[2], Majid Khabbazian[3], and Calvin Newport[4,⋆]

[1] Ben-Gurion University, Beersheba, Israel
[2] National University of Singapore, Singapore
[3] University of Winnipeg, Winnipeg, Canada
[4] MIT CSAIL, Cambridge, USA

Abstract. This paper addresses two primary questions: (i) How much faster can we disseminate information in a large wireless network if we have multiple communication channels available (as compared to relying on only a single communication channel)? (ii) Can we still disseminate information reliably, even if some subset of the channels are disrupted? In answer to the first question, we reduce the cost of broadcast to $O(\log \log n)$ rounds/hop, approximately, for sufficiently many channels. We answer the second question in the affirmative, presenting two different algorithms, while at the same time proving a lower bound showing that disrupted channels have unavoidable costs.

1 Introduction

This paper addresses two primary questions: (i) How much faster can we disseminate information in a large wireless network if we have multiple communication channels available (as compared to relying on only a single communication channel)? (ii) Can we still disseminate information reliably, even if some subset of the channels are disrupted? In answer to the first question, we reduce the cost of broadcast to $O(\log \log n)$ rounds/hop, approximately, for sufficiently many channels. We answer the second question in the affirmative, presenting two different algorithms, while at the same time proving a lower bound showing that disrupted channels have unavoidable costs.

Multi-channel Networks. In more detail, we study the multihop broadcast problem in the t-disrupted radio network model [11–14, 17, 26, 30, 31]. This model describes a synchronous multihop radio network, and assumes that in each round, each process chooses 1 out of $\mathcal{C}$ available communication channels to participate on. Simultaneously, an adversary selects, at each process, up to $t < \mathcal{C}$ channels to locally *disrupt*, preventing communication. In this study, we also equip processes with receiver collision detectors, but assume that disruption is indistinguishable from collisions.

⋆ Research supported in part by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Microsoft, Deutsche Telekom, VeriSign .Com grant, Israeli Defense Secretary (MFAT), US Air-Force and Rita Altura Trust Chair in Computer Sciences, NUS (FRC) R-252-000-443-133, and Natural Sciences and Engineering Research Council of Canada.

As detailed in [11, 26], the adversary in the t-disrupted model does not represent a literal adversarial device; it cannot spoof messages or reactively jam a broadcast (i.e., scan the channels to discover a broadcast in progress, then jam the remainder of transmission). The adversary instead incarnates the unpredictable message loss that plagues real radio network deployments. This message loss has many (non-malicious) causes, including: unrelated protocols using the same unlicensed spectrum band, time-varying multipath effects, and electromagnetic interference from non-radio devices, such as microwaves. The goal of the t-disrupted model is two-fold: (1) to improve *efficiency*: most real radio network protocols have access to multiple communication channels,[1] and therefore theoretical algorithms should enjoy this same advantage; and (2) to improve *robustness*: a protocol proved correct in a model with unpredictable message loss is a protocol more likely to remain correct in a real deployment, where such loss is often unavoidable.

Results: No Disruption. We start by showing that adding communication channels makes broadcast more efficient, yielding $O(\log\log n)$ rounds/hop in a network of diameter $D > \log n$ with $\Theta(\log n)$ channels. In more detail, in the setting with no disruption ($t = 0$), we present a randomized algorithm that solves broadcast in $O((D + \log n)(\log \mathcal{C} + \frac{\log n}{\mathcal{C}}))$ rounds, w.h.p. Notice, for a single channel ($\mathcal{C} = 1$), our algorithm has the same running time as the canonical Bar-Yehuda et. al algorithm [3], but as the number of channels increases so does our algorithm's performance advantage. This comparison however, is not exact, as unlike [3], we assume collision detection. With this in mind, we prove a lower bound $\Omega(D + \frac{log^2 n}{\mathcal{C}})$ rounds for broadcast algorithms in our collision detector-equipped model. It follows that for $\mathcal{C} = \Omega(\log n)$: our algorithm is within a factor of $O(\log\log n)$ of optimal, and for sufficiently small D, it is strictly more efficient than the best possible single-channel algorithm.

The key insight of this algorithm is the following: At a high-level, standard single-channel broadcast algorithms, such as [3], require processes to sequentially test $\log n$ broadcast probabilities, exponentially distributed between $1/n$ and $1/2$. The idea is that for every process with transmitting neighbors, one of these probabilities will match what is required for the message to be received. Our algorithm, by contrast, leverages multiple communication channels to test multiple probabilities *in parallel*, allowing processes to hone in on the correct probabilities more efficiently.

While it may not be surprising that some speed-up is possible using multiple channels, it is non-trivial to determine exactly what is feasible for two reasons. First, the multiple communication channels can only speed up one part of the algorithm (i.e., the contention resolution); it cannot speed-up the time to relay the message over long distances. Second, the multiple channels cannot all be used in parallel by any one processor, as each has only one transceiver. Thus, the "obvious" solutions, e.g., multiplexing the single-channel protocol over multiple channels, are not applicable. If $\log n$ channels could be used in parallel, we could readily achieve a rounds-per-hop cost of $O(1)$; that we can still achieve $O(\log\log n)$ rounds-per-hop with only one transceiver, is, perhaps, surprising.

[1] The 802.11 b/g network protocols [1], for example, divide the shared 2.4 Ghz band into 13 channels, while Bluetooth [4] divides the same band into 79 channels.

Results: t-Disruption. Having showing that additional communication channels improves *efficiency*, we next turn our attention to showing that they also improve *robustness*. We now assume disruption (i.e., $t > 0$) and that processes have access to a common source of random bits. We argue that this latter assumption is often justified in practice, as most radio network deployments require devices to be configured with a common *network id*, and a hash of this id can provide a seed for a pseudo-random bit generator.[2]

In this setting, we present a randomized algorithm that solves broadcast in $O((D + \log n)(\frac{\mathcal{C} \log \mathcal{C} \log \log \mathcal{C}}{\mathcal{C}-t} + \frac{\log n}{\mathcal{C}-t}))$ rounds, w.h.p., where t is the upper bound on disrupted channels. Notice, for t up to a constant factor of $\mathcal{C}$, this algorithm performs only a factor of $O(\log \log \mathcal{C})$ slower than the no disruption case. In other words, even with lots of disruption, our multi-channel algorithm still outperforms the best possible single channel solution in many cases, and is more efficient than the canonical single channel algorithm of [3]. The key insight of this algorithm is that we replace the broadcast and receive primitives used in the no disruption case with *simulated* versions. These simulated broadcasts and receives use the common randomness to generate coordinated random frequency hopping patterns. These patterns are used to evade adversarial disruption with sufficient probability for the original no disruption arguments to still apply.

Lastly, we consider the case with disruption and *no* common randomness. We describe a randomized algorithm that solves broadcast in this setting in $O((D+\log n)\frac{\mathcal{C}t}{\mathcal{C}-t} \cdot \log(\frac{n}{t}))$ rounds, w.h.p. Notice, for large t, this algorithm now performs slightly worse than [3], but this is arguably still a reasonable price to pay for the added robustness. We conclude by showing this price to be not just reasonable, but also be necessary. In more detail, we prove a lower bound of $\Omega((D + \log n)\frac{\mathcal{C}t}{\mathcal{C}-t})$ rounds to solve broadcast in this setting.

Related Work. We use the terminology *multihop broadcast* to describe the problem addressed in this paper, as we want to clearly separate it from the *local broadcast* problem we solve as a subroutine. Previous work on this problem, however, has used both *reliable broadcast* (e.g., [19]) and *broadcast* (e.g., [3]) to refer to the same problem. All terms describe the same goal of disseminating a message from a single distinguished source to every process in a radio network.

Theoretical concern with broadcasting in radio networks began with the investigation of centralized solutions. Chlamtac and Kutten [5] opened the topic by proving the calculation of optimal broadcast schedules to be NP-hard, Chlamtac and Weinstein [6] followed with a polynomial-time algorithm that guaranteed schedule lengths of size $O(D \log^2 n)$, and Alon et al. proved the existence of constant diameter graphs that require $\Omega(\log^2 n)$ rounds [2]. An oft-cited paper by Bar Yehuda et al. [3] introduced the first *distributed* solution to broadcast, launching a long series of papers investigating distributed solutions under different model assumptions; c.f., [7–10, 22]. The algorithm in [3] assumes no topology knowledge or collision detection, and solves broadcast in

[2] Note, if the adversary in the t-disrupted model represented an actual adversarial device, we would have to worry about keeping such information secure. But as explained previously, this adversary is an abstraction of the diverse, and hard to predict interference sources that plague real networks, and does not represent behavior with malicious intent.

$O((D + \log n) \log(n))$ rounds, w.h.p. In later work [10, 21], this bound was improved to $O((D+\log n) \log(n/D))$, which performs better in graphs with large diameters. For the assumption of no topology knowledge, these broadcast bounds can be considered the best known. In centralized setting, the optimal result $\Theta(D + \log^2 n)$ in undirected multi-hop networks is given in [16] and [23].

Our algorithm for the no disruption setting matches the Bar-Yehuda algorithm for the case where $\mathcal{C} = 1$, and performs increasingly better as we increase the number of channels. Its comparability with the bound of [10, 21] depends on the diameter. Our model, however, unlike the model in [3, 10, 21], assumes receiver collision detection, so these comparisons are not exact. (The $O(D \log n)$-time broadcast algorithm of [29], by contrast, *does* assume collision detection, but a direct comparison is foiled in this case because the model of [29] constrains the communication graph to be growth-bounded, whereas our model, as in the canonical results referenced above, works for arbitrary graphs.) This motivates the $\Omega(D + \frac{\log^2 n}{\mathcal{C}})$ lower bound we prove in Section 6 for solving broadcast in our model. Notice, this implies that the best possible single-channel broadcast algorithm in our model requires $\Omega(D + \log^2 n)$ rounds. For $\mathcal{C} = \Omega(\log n)$, and sufficiently small D, our no disruption algorithm is strictly more efficient. In Section 4, we show that even if we introduce significant disruption, if we assume a common source of randomness we still outperform the best possible single channel solution in many cases.

Koo [19] considered broadcast in a model that assumed a single channel and Byzantine failures, which, due to their ability to spoof messages, are arguably more challenging than the disruption faults considered in our work. The corrupt processes in this model, however, could not disrupt communication. In later work, Koo, now collaborating with Bhandari, Katz, and Vaidya [20], extended the model to allow for a bounded number of collisions. Their focus was on feasibility (i.e., for what amount of corruptions is broadcast still solvable) not time complexity. Drabkin et al. [15] and Pelc and Peleg [27] both studied broadcast in radio network models that assume a single channel and probabilistic message corruption. Finally, in recent work, Richa et al. [28] considered efficient MAC protocols in a single channel, multihop radio network, with an adversary that can cause a bounded amount of communication disruption.

2 Model

We model a synchronous multihop radio network with multiple communication channels, symmetric communication links, receiver collision detection, and adversarial disruption. In the following, for integer $x > 1$, let $[x] = \{1, ..., x\}$, and assume $\log$ denotes the base-2 logarithm. Fix an undirected graph $G = (V, E)$, with diameter D, where the vertexes in V correspond to the $n > 1$ processes in the network, which we uniquely label from $[n]$. We assume processes know n. To simplify notation we also assume that n is a power of 2. In this paper, when we denote a property holds *with high probability* (w.h.p.), we assume a probability of at least $1 - \frac{1}{n^x}$, for some sufficiently large positive integer x. Fix a set $[\mathcal{C}]$ of communication channels for some integer $\mathcal{C} \geq 1$, and a known upper bound on disruption, t, $0 \leq t \leq \mathcal{C}$. Executions in our model proceeds in synchronous rounds labeled $1, 2, \ldots$. Because we study broadcast problems, we assume processes can receive a message *from* and output a message *to* the environment, during each round. All processes start in round 1, but following the standard assumption made in the study of multihop broadcast (e.g., [3]), we assume no process can broadcast before it receives a message, either from another process or the environment.

To model disruption, we use the *t-disrupted model*, which was introduced in [13], and has since been extensively studied in the context of both single hop and multihop radio networks [11–14, 17, 26, 30, 31]. (See [11] for a good overview of this model and results.[3]) In the *t-disrupted model*, in each round r, an adversary chooses, for each process i, a set $disp(i, r)$ of up to t channels to *disrupt*. The adversary can use the history of the execution through round $r - 1$, as well as the process definitions, in deciding $disp(i, r)$. It does not, however, have advance knowledge of the random choices made in r. We consider two cases for the random choices: (i) *common randomness*, where processes can access a common source of random bits in each round, and (ii) *no common randomness*, case where the bits are independent at each process. Next, each process i chooses a channel $c \in [\mathcal{C}]$ on which to participate, and decides whether to *broadcast* or *receive*. If i broadcasts it receives nothing. If i receives, three behaviors are possible: (1) if no neighbor of i in G broadcasts on c in r and $c \notin disp(i, r)$, i detects silence, indicated by $\perp$; (2) if exactly one neighbor j of i in G broadcasts on c in r, and $c \notin disp(i, r)$, i receives j's message; (3) if two or more neighbors of i in G broadcast on c in r, or $c \in disp(i, r)$, i detects a collision, indicated by $\pm$. (That is, receiving on a disrupted channel is indistinguishable from detecting a collision.) Notice, i learns nothing about the activities of processes on other channels during this round.

The Multihop Broadcast Problem. Our goal in this paper is to define bounds for the *multihop broadcast problem*, which is defined as follows: At the beginning of round 1, a single *source* process is provided a message m by the environment. We say an algorithm *solves the multihop broadcast problem in r rounds* if and only if every process outputs m by round r, w.h.p.

The Local Broadcast Problem. In this paper, following the approach of [18], we decompose multihop broadcast into first solving local broadcast, and then using the construction presented in [18] to transform this local solution into a global one.

In more detail, the *T_A-local broadcast problem*, for positive integer T_A, assumes that the environment injects a message m at arbitrary processes at arbitrary times, and that every process that receives the message from the environment must eventually output *ack*. We say an algorithm *solves the T_A-local broadcast problem* if and only if the following hold: (a) If some process i receives the message from the environment in round r and outputs *ack* in round $r' \geq r$, then all neighbors of i output the message by round r', w.h.p. (b) We say a process is *active* in a given round r if it received the message from the environment in some round $r' \leq r$, and it has not yet output *ack* by the beginning of r. Given any interval of T_A rounds, if process i has a neighbor that is active in every round of the interval, then i outputs the message by the end of the interval, with *constant probability.*

[3] This model has been called many different names. Originally [13] it was unnamed; later works[11, 14] called it the *disrupted radio network* model; it was only in more recent work [26] that the more descriptive name of *t-disrupted* was introduced.

Transforming Local Broadcast to Multihop Broadcast. The following theorem, which follows from Theorem 7.8 of [18], reduces the problem of multihop broadcast to local broadcast:[4]

Theorem 1 (Theorem 7.8 of [18]). *Given an algorithm that solves the T_A-local broadcast problem, we can construct an algorithm that solves the multihop broadcast problem in $O((D + \log n)T_A)$ rounds.*

3 Upper Bound for No Disruption

We begin with an algorithm for the case with no disruption (i.e., $t = 0$), that solves multihop broadcast in $O((D + \log n)(\log \mathcal{C} + \frac{\log n}{\mathcal{C}}))$ rounds. For $\mathcal{C} = 1$, this running time matches the canonical broadcast algorithm of Bar-Yehuda et al. [3], but as the number of channels increases so does our performance advantage. In Section 6, we will prove that for sufficiently large $\mathcal{C}$, this is within a $O(\log \log n)$ factor of optimal.

As described in Section 2, our approach is to first solve the *local* broadcast problem, then apply Theorem 1 to generate our *global* solution. Our algorithm only makes use of up to $\log n$ channels, so in this section we assume, w.l.o.g., $\mathcal{C} \leq \log n$.

Intuition. The key insight of our protocol is to trade channel diversity for time complexity. Most existing broadcast algorithms (e.g., [3]) described at a high level, have processes sequentially test $\log n$ different broadcast probabilities exponentially distributed between $1/n$ and $1/2$. For each process waiting to receive a message from transmitting neighbors, one of these probabilities should sufficiently reduce the contention and hence match what is needed to ensure that the message is delivered. Our algorithm, by contrast, leverages multiple channels to test multiple probabilities in parallel, gaining efficiency.

Our local broadcast algorithm consists of two subroutines: SEARCH and LISTEN. During, SEARCH, processes assign an exponential distribution of probabilities to the channels (captured by $schan$ in our algorithm description). A receiving process can then do a binary search over the channels (with silence indicating the probability is *too low*, and a collision indicating *too high*), to find the probability that best matches the number of transmitting neighbors. (This search is what necessitates receiver collision detection in our model.) If $\mathcal{C} \leq \log n$, however, then this SEARCH subroutine identifies only a rough range of $\log n/\mathcal{C}$ probabilities, in which is included the right probability

[4] Formally, the local broadcast problem described above is a simplified presentation of the *Abstract MAC Layer* formalism first introduced in [24]. The result cited from [18] provides an implementation of multihop broadcast that uses a probabilistic Abstract MAC Layer implementation. Our definition of local broadcast simplifies the Abstract MAC Layer definition down to only the properties needed to apply the transformation in [18]. In more detail, receiving a message m from the environment in our model corresponds to $bcast(m)$ in the Abstract MAC Layer, and outputting the message corresponds to calling $recv(m)$. In addition, the T_A parameter corresponds to $f_{prog}(\Delta)$, the constant probability of the T_A property holding corresponds to $1-\epsilon_{prog}$, and the high probability of all neighbors eventually outputting the message corresponds to $1 - \epsilon_{ack}$. We do not define an equivalent of f_{ack} or f_{rcv}, as neither are used in the transformation. We point the interested reader to [18] for more details.

for actually receiving a message. During the LISTEN subroutine, transmitting processes cycle through the different probabilities assigned to each channel (captured by $lchan$ in our algorithm description).[5] In both subroutines, care must be taken to account for the fact that many processes are both transmitters and receivers: a problem we solve by having processes choose a role with probability $1/2$.

Algorithm Description. The local broadcast algorithm has all processes alternate between executing the SEARCH and LISTEN subroutines presented in Figure 1, starting in round 1. Each call to SEARCH returns a candidate channel c_1, and the following call to LISTEN is made with $channel = c_1$. On receiving a message msg from the environment, a process sets $m \leftarrow msg$, and continues to try to transmit the message for the subsequent $AMAX$ calls to both subroutines, starting with the next call to SEARCH. After these $AMAX$ calls it outputs *ack*. In all other rounds, it sets $m \leftarrow \perp$. (Note, as required by our model, processes do not broadcast until they first receive a message.)

Constants Used in Algorithm. Let $k = \lceil \frac{\log n}{\mathcal{C}} \rceil$. The constant k represents the (approximate) number of probabilities assigned to each channel. Let $p_c = 1/2^{k(c-1)+1}$, for $c \in [\mathcal{C}]$. The function $schan()$ returns channel $c \in [\mathcal{C}]$ with probability p_c, and the *null* channel 0 with the sum of the remaining probability: $1 - \sum_{c \in [\mathcal{C}]} p_c$. That is, $schan$ chooses channels using an exponential probability distribution.

Next, we define a family of functions $lchan(r)$, for $r \in \{1, ..., k+3\}$. Intuitively, $lchan$ partitions the $\log n$ probabilities from $\{\frac{1}{n}, \frac{2}{n}, ..., \frac{1}{2}\}$ among the $\mathcal{C}$ channels. This means that k, defined above as $\lceil \frac{\log n}{\mathcal{C}} \rceil$, describes the number of probabilities in each channel partition. For each index passed to $lchan$, it assigns channels one of the probabilities from their partition, and then randomly selects a channel based on this distribution. The function $lchan(r)$ is defined as follows: if $r = 1$, it returns channel 1 with probability 0; if $r > k + 1$, it returns channel $\mathcal{C}$ with probability 0; if $r = k + 3$ and $k = 1$, it returns channel $\mathcal{C} - 1$ with probability 0; for all other r and c pairs, it returns channel c with probability $(2p_c)/2^{r-1}$. As with $schan()$, it returns the *null* channel 0 with the sum of the remaining probabilities for the given r value. The function $lchan$is defined for more than k values (i.e., $k + 3$ instead of k) because, to simplify the proof later, it helps if in addition to using every probability in a given channel's partition, we also use a constant number of probabilities that have been assigned to neighboring channels.

Finally, let $SMAX = 2(\lceil \log(\mathcal{C}) \rceil + 1)$, $LMAX = \lceil \frac{\log n}{\mathcal{C}} \rceil + 3$, and $AMAX = \Theta(\log n)$, where the constants are defined in our main theorem proof.

Correctness Proof. We prove that each pair of calls to SEARCH and LISTEN receives a message with constant probability, assuming there is a message to be received. We also prove that over AMAX calls to these subroutines, a message is received w.h.p. It follows that we solve T_A-local broadcast problem for $T_A = O(SMAX + LMAX)$, which when combined with Theorem 1 yields an algorithm that solves multihop broadcast problem in $O((D + \log n)(\log \mathcal{C} + \frac{\log n}{\mathcal{C}}))$ rounds.

[5] To make the probabilities work in our proofs, transmitters also try, for each channel, a constant number of probabilities from the neighboring channels. This is why $lchan$ cycles through $\log n/\mathcal{C} + O(1)$ different probability assignments, not just $\log n/\mathcal{C}$.

```
**SEARCH**$(m)$
$c_1 \leftarrow 1; c_2 \leftarrow \mathcal{C}; count \leftarrow 1$
while $count \leq SMAX$
    $phase \leftarrow random(broadcast, listen)$
    if $(phase = broadcast)$ and $(m \neq \bot)$
        $b_c \leftarrow schan()$
        **bcast**$(m, b_c)$
    else if $(phase = listen)$ and $(c_1 \neq c_2)$
        $channel \leftarrow \lceil c_1 + (c_2 - c_1)/2 \rceil$
        $rmsg \leftarrow$ **recv**$(channel)$
        if $(rmsg = \bot)$ then $c_2 \leftarrow channel - 1$
        else $c_1 \leftarrow channel$
    $count \leftarrow count + 1$
```

```
**LISTEN**$(m, channel)$
$count \leftarrow 1$
while $count \leq LMAX$
    $phase \leftarrow random(broadcast, listen)$
    if $(phase = broadcast)$ and $(m \neq \bot)$ then
        $b_c \leftarrow lchan(count)$
        **bcast**$(m, b_c)$
    else
        $rmsg \leftarrow$ **recv**$(channel)$
        if $(rmsg \neq \bot)$ and $(rmsg \neq \pm)$ then
            **output**$(rmsg)$
    $count \leftarrow count + 1$
```

Fig. 1. The SEARCH and LISTEN subroutines called by our local broadcast solution. $SMAX = \Theta(\log \mathcal{C})$ and $LMAX = \Theta(\log n/\mathcal{C})$.

To begin the proof, fix a process i and a call to SEARCH. Let I, $|I| \leq \Delta$, be the set of *active* neighbors of i during this call—that is, the neighbors of i with a message to send (i.e., $m \neq \bot$ in their call to SEARCH). We say a call to SEARCH is *valid* for this process i if and only if these following three conditions hold: (1) at the conclusion of the subroutine, $c_1 = c_2$; (2) for each **recv**(c), if $p_c|I| \leq \frac{1}{2}$, it returns $\bot$; and (3) for each **recv**(c), if $p_c|I| \geq 4$, it does not return $\bot$. (Otherwise, if a call to SEARCH is invalid, it may return a channel with too much or too little contention.) We prove this occurs with constant probability:

Lemma 1. *The call to SEARCH is valid with constant probability.*

Proof (Proof (sketch)). To prove the first condition of validity we must show that SEARCH sets $phase \leftarrow listen$ at least $\gamma \geq (\lceil \log(\mathcal{C}) \rceil + 1)$ times. Since this occurs according to a binomial distribution, with median $\frac{1}{2} \cdot SMAX \geq \gamma$, we conclude that with probability at least $\frac{1}{2}$ the SEARCH completes with $c_1 = c_2$.

To prove the second condition, let $\mathcal{L}$ contain every channel c such that i receives on c and assume $p_c|I| \leq \frac{1}{2}$. For a given $c \in \mathcal{L}$, the condition holds with probability $(1 - \frac{1}{2}p_c)^{|I|}$, and hence by a union bound, the condition holds over all relevant rounds with probability at least $1/2$. We prove the third condition in a similar manner, concluding that the condition holds over all relevant rounds with probability at least 0.8.

We now show that if process i's call to SEARCH is valid then, with constant probability, process i will receive a message during the subsequent call to LISTEN (assuming, of course, $|I| > 0$).

Lemma 2. *Suppose process i's call to SEARCH is valid and $|I| > 0$. Then, process i will receive a message during the subsequent LISTEN subroutine, with constant probability.*

Proof (Proof (sketch)). Let c be the channel returned by the call to SEARCH.

We consider two cases for the size of I. In the first case, assume $|I| = 1$. Here, $p_{c'}|I| \leq \frac{1}{2}$ for every channel $c' > 1$. Since we assume SEARCH was valid (with constant probability), every call to **recv** during the subroutine would return $\perp$. It follows that LISTEN executes on channel $c = 1$, where process i will receive a message with probability at least $\frac{1}{2} \cdot \frac{1}{2} \cdot p_1 = \frac{1}{8}$.

For the second case, assume $|I| > 1$. Let p_{min} be the smallest non-0 probability assigned to channel c in all $k + 3$ calls to $lchan$ in the listen phase, and let p_{max} be the largest probability. We can then bound both p_{max} and p_{min}: $p_{max}|I| \geq 1$ and $p_{min}|I| \leq 2$.

By definition, $p_{min} \leq p_{max}$. Combined, we conclude that there must exists a probability p', among those assigned to channel c by $lchan$ during LISTEN such that $1 \leq p'|I| \leq 2$. Consider the LISTEN round during which p' is assigned to channel c by $lchan$. During this round, process i will receive a message with probability $p_{rcv} \geq \frac{1}{2}p''|I|(1 - p'')^{|I|-1}$, where $p'' = \frac{p'}{2}$ is the probability that a process broadcasts in channel c in that round, and the first $\frac{1}{2}$ bounds the probability that i receives. Note that $\frac{1}{2} \leq p''|I| \leq 1$ and $p'' \leq \frac{1}{2}$. We now simplify p_{rcv}: $p_{rcv} = \frac{1}{2}p''|I| \left(1 - p''\right)^{|I|-1} \geq \frac{1}{2}p''|I| \left(1 - p''\right)^{|I|}$. This later term is greater than or equal to $\frac{1}{2}p''|I| \left(\frac{1}{4}\right)^{p''|I|} \geq \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{8}$. Notice, the third step uses our above-stated fact that $p'' \leq \frac{1}{2}$, and the fourth step uses the other above-stated fact that $\frac{1}{2} \leq p''|I| \leq 1$.

We can now prove that the algorithm solves the local broadcast problem.

Lemma 3. *The algorithm solves the $2(SMAX + LMAX)$-local broadcast problem.*

Proof. By Lemmas 1 and 2, we know the algorithm satisfies property (b) of the local broadcast problem, for $T_A = 2(SMAX + LMAX)$ (the factor of 2 accounts for the case that a message arrives after SEARCH has begun, necessitating we wait until the next call to SEARCH begins before the process begins trying to send the message). To show the algorithm satisfies property (a), assume that some process i receives the message from the environment for the first time at some round r. Let j be a neighbor of i. By our above argument, over the next $AMAX$ pairs of calls to SEARCH and LISTEN, j will receive the message from i (or another neighboring process) with some constant probability p. Process j therefore fails to receive the message in all $AMAX$ pair of calls, with probability no greater than $(1 - p)^{AMAX} \leq e^{-p \cdot AMAX}$. Because p is constant and $AMAX = O(\log n)$, for sufficiently large constant factors, this failure with probability no more than $\frac{1}{n^{x+1}}$, for any positive constant x. By a union bound over the $O(n)$ neighbors of i, property (a) holds w.h.p., as needed.

Given Lemma 3, we can now apply Theorem 1 to derive our final result:

Theorem 2. *We can construct an algorithm that solves the multihop broadcast problem with no disruption ($t = 0$) in $O((D + \log n)(\log \mathcal{C} + \frac{\log n}{\mathcal{C}}))$ rounds.*

4 Upper Bound for Disruption and Common Randomness

In this section, we assume that channels may be disrupted (i.e., $t > 0$) and that processes have access to a common source of randomness. We present an algorithm that solves the

```
sim-bcast_γ(m, c):
$simcount \leftarrow 1$
while $simcount \leq \gamma$
    $\psi \leftarrow$ a channel permutation generated with
    common source of randomness for this round.
    bcast$(m, \psi(c))$
    $simcount \leftarrow simcount + 1$
```

```
sim-recv_γ(c)
$rmsg \leftarrow \pm$; $simcount \leftarrow 1$
while $simcount \leq \gamma$
    $\psi \leftarrow$ a channel permutation generated with
    common source of randomness for this round.
    $m \leftarrow$ recv$(\psi(c))$
    if $m \neq \pm$ then $rmsg \leftarrow m$
    $simcount \leftarrow simcount + 1$
return $rmsg$
```

Fig. 2. The simulated broadcast and receive functions that replace the **bcast** and **recv** functions of the no disruption algorithm (Figure 1) to produce a local broadcast algorithm for the setting with disruption and a common source of randomness. For SEARCH, $\gamma = \Theta(\frac{\mathcal{C}}{\mathcal{C}-t} \log\log \mathcal{C})$, and for LISTEN, $\gamma = \Theta(\frac{\mathcal{C}}{\mathcal{C}-t})$.

multihop broadcast problem in $O((D + \log n)(\frac{\mathcal{C}\log\mathcal{C}\log\log\mathcal{C}}{\mathcal{C}-t} + \frac{\log n}{\mathcal{C}-t}))$ rounds, where t is the known upper bound on disrupted channels. Therefore, for even large amounts of disruption (i.e., for any t up to a constant factor of $\mathcal{C}$) our disruption-tolerant protocol performs only a factor of $O(\log\log \mathcal{C})$ slower than our no disruption protocol from Section 3. This means that for sufficiently large $\mathcal{C}$, we still outperform the best possible single channel solution in many cases, and are more efficient than the canonical single channel algorithm of [3]. It follows that common randomness is a potent weapon against disruptive interference.

Intuition. Our approach is to extend the no disruption algorithm from Section 3. In more detail, we replace the broadcast and received primitives of the no disruption protocol with disruption-tolerant versions that use coordinated frequency hopping (specified by the common random bits) to evade disruption. We show that the new **sim-recv** subroutine outputs the same value as its no disruption counterpart (i.e., the **recv** subroutine) with *just enough* probability to ensure that our analysis still applies. Note that the new subroutines call the **bcast** and **recv** functions several times, but not so much that the running time becomes unwieldy.

Algorithm Description. Our local broadcast algorithm replaces each call to **bcast** and **recv** in the no disruption subroutines from Figure 1, with calls to *simulated* broadcasts and receives that use multiple rounds to evade disruption. In more detail, in our modified version of the SEARCH subroutine from Figure 1, which we call DSEARCH, we replace each call to **bcast**(m, b_c) with a call to **sim-bcast**$_{\gamma_S}(m, b_c)$, and each call to **recv**$(channel)$ with a call to **sim-recv**$_{\gamma_S}(channel)$, where **sim-bcast** and **sim-recv** are defined in Figure 2, and $\gamma_S = \Theta(\frac{\mathcal{C}}{\mathcal{C}-t} \log\log \mathcal{C})$. For the modified LISTEN subroutine, which we call DLISTEN, we do the same replacement of **bcast** and **recv** with **sim-bcast** and **sim-recv**, substituting $\gamma_L = \Theta(\frac{\mathcal{C}}{\mathcal{C}-t})$ for γ_S. For any round r of an execution, we assume that every process generating the random channel permutation ψ during r, will generate the same permutation, using the common randomness.

Correctness Proof. We begin by bounding the probability that our simulated bcast and receives behave the same as if we were in the no disruption setting. This claim follows primarily from the fact that the probability that the adversary disrupts every channel in $\psi(c)$ in the relevant round is $O(1/\log \mathcal{C})$.

Lemma 4. *Suppose process i calls* **sim-recv** *during some round of DSEARCH, and all of i's neighbors also call either* **sim-bcast** *or* **sim-recv** *during this same round. With probability at least $1 - O(\frac{1}{\log \mathcal{C}})$,* **sim-recv** *will return i the same value as if these same processes had called* **bcast** *and* **recv***, with the same parameters, in the setting where $t = 0$.*

If we replace γ_S with γ_L, we can show a similar result for DLISTEN, this time with constant probability:

Lemma 5. *Suppose process i calls* **sim-recv** *during some round of DSEARCH, and all of i's neighbors also call either* **sim-bcast** *or* **sim-recv** *during this same round. With constant probability,* **sim-recv** *will return i the same value as if these same processes had called* **bcast** *and* **recv***, with the same parameters, in the setting where $t = 0$.*

We now show, much as in Section 3, that the local broadcast performs well:

Lemma 6. *The algorithm solves the $2(SMAX \cdot \gamma_S + LMAX \cdot \gamma_L)$-local broadcast problem.*

Given Lemma 6, we apply Theorem 1 to derive our final result regarding multihop broadcast:

Theorem 3. *We can construct an algorithm that solves the multihop broadcast problem with common randomness in $O((D + \log n)(\frac{\mathcal{C} \log \mathcal{C} \log\log \mathcal{C}}{\mathcal{C} - t} + \frac{\log n}{\mathcal{C} - t}))$ rounds.*

5 Upper Bound for Disruption and No Common Randomness

In this section, we assume disruption and no common source of randomness. We present an algorithm that solves multihop broadcast in $O((D + \log n)\frac{\mathcal{C}t}{\mathcal{C}-t} \log(\frac{n}{t}))$ rounds. In Section 6, we prove this to be within a factor of $O(\log(\frac{n}{t}))$ of optimal. Unlike the common randomness case, here we actually perform (slightly) worse than the single channel algorithm of [3] (at least, for large t). This difference, however, is bounded by a factor of $O(\log n)$, which is arguably still a reasonable price to pay for the increased robustness. In the following, we assume w.l.o.g. that $\mathcal{C} < 2t$.

Intuition. There are three basic challenges to overcome: First, because some t channels are disrupted, processes must attempt to communicate on more than t channels, and to avoid the disruption, the communication must be randomized. Second, since the processes have no source of common randomness, the random channel selection potentially delays the receivers from finding the broadcasts. Third, processes still have to solve the problem of contention, i.e., the fact that many broadcasters may be competing to send a message. To overcome these problems, we have processes repeatedly choose channels uniformly at random, cycling through the $\log n$ broadcast probabilities that are exponentially distributed between $1/n$ and $1/2$.

Algorithm Description. Our local broadcast algorithm works as follows. First, the execution is divided into epochs of length $\lceil \log n/\mathcal{C} \rceil$. If a message is injected at a process v in some round r, then process v waits until the beginning of the next epoch before trying to disseminate the message. We say that a process that has received message m by the first round of some epoch e, but has not yet returned an acknowledgment for m, *participates* in epoch e. In each round of an epoch, each participating process decides whether to broadcast and on which channel. In particular, in round r, a participating process v broadcasts with probability $1/2^r$; it chooses channel $c \in [\mathcal{C}]$ with probability $1/\mathcal{C}$. Every process u that is not broadcasting a message chooses a channel on which to listen with the same uniform probability $1/\mathcal{C}$. A process v returns an acknowledgment when it has participated for $\Theta((\mathcal{C}^2 \log n)/(\mathcal{C} - t))$ epochs.

Correctness. We argue that this protocol solves T_A-local broadcast for $T_A = O((\mathcal{C}^2 \log n/C)/(\mathcal{C} - t))$. We first argue that if process u has a participating neighbor in epoch e, then by the end of the epoch, it receives the message with constant probability:

Lemma 7. *Let u be a process that has not received the message m prior to epoch e. Let V be the set of neighbors of u participating in epoch e, and assume that $|V| > 0$. Then with probability at least $(\mathcal{C} - t)/(32\mathcal{C}^2)$, u receives message m by the end of epoch e.*

Proof (Proof (sketch)). Process u receives the message m in a round r if the following three conditions are satisfied: (a) there is exactly one process in $v \in V$ that broadcasts in round r; (b) the channel selected by v is not disrupted in round r; and (c) process u chooses to listen on the same channel on which v broadcasts in round r. Consider round $r = \lceil \log |V| \rceil$ in epoch e. We now bound the probability that these three events occur.

Let $c \in \mathcal{C}$ be the channel chosen by u in round r of epoch e. With probability $(\mathcal{C} - t)/\mathcal{C}$ we observe that c is not disrupted. We calculate the probability that exactly one participating process in V broadcasts on channel c: $\sum_{v \in V} \frac{1}{\mathcal{C}2^r} \left(1 - \frac{1}{\mathcal{C}2^r}\right)^{|V|-1} \geq 1/(32\mathcal{C})$. Thus, with probability $(\mathcal{C} - t)/(32\mathcal{C}^2)$, process u receives the message by the end of the epoch.

From this we conclude that the protocol solves the T_A-local broadcast problem:

Lemma 8. *The specified protocol solves the T_A-local broadcast problem for $T_A = O(\frac{\mathcal{C}^2 \log(n/\mathcal{C})}{(\mathcal{C}-t)})$.*

Proof. First, we argue that every process with active neighbors receives the message within time T_A with constant probability. Consider a process u. By Lemma 7, during each epoch in which a neighbor participates, u receives the message with probability $(\mathcal{C} - t)/(32\mathcal{C}^2)$. Thus, over $(32\mathcal{C}^2/(\mathcal{C} - t))$ epochs, process u receives the message with constant probability. An active neighbor may not participate for the first epoch when it receives the message, and from this we conclude that if $T_A = [(32\mathcal{C}^2/(\mathcal{C} - t)) + 1] \log(n/\mathcal{C})$, then u receives the message as required.

Next, we argue that when a node sends an acknowledgment, every neighboring process has received the message. Specifically: in each epoch, each neighbor receives the message with constant probability. Thus within $O(\log n)$ epochs, every neighbor has received the message with high probability, as required.

Since $t < \mathcal{C} \leq 2t$, we can apply Theorem 1 to conclude:

Theorem 4. *We can construct an algorithm that solves the multihop broadcast problem without common randomness in* $O((D + \log n)(\frac{\mathcal{C}t}{\mathcal{C}-t}) \log{(\frac{n}{t})})$ *rounds.*

6 Lower Bounds

We begin by showing that the $O((D + \log n)(\log \mathcal{C} + \frac{\log n}{\mathcal{C}}))$-time broadcast algorithm from Section 3 is (almost) tight for sufficiently large $\mathcal{C}$, by proving a $\Omega(D + \frac{\log^2 n}{\mathcal{C}})$ lower bound for solving broadcast in this setting. (In more detail, for $\mathcal{C} = \Omega(\log n)$, the upper bound is within a factor of $O(\log \log n)$ of the lower.)

Theorem 5. *For any* $D \leq n/2$*: every multihop broadcast algorithm requires* $\Omega(D + \frac{\log^2 n}{\mathcal{C}})$ *rounds.*

Proof. We first note that we can simulate any protocol for a network with $\mathcal{C} > 1$ in a network where $\mathcal{C} = 1$. In more detail, we use $\mathcal{C}$ rounds in the single channel network to simulate each round from the multi-channel network, with each simulation round being dedicated to a different channel. It follows that if $f(n)$ is a lower bound for multihop broadcast in a network where $\mathcal{C} = 1$, then $f(n)/\mathcal{C}$ is a lower bound for networks with larger $\mathcal{C}$. The question remains what lower bounds apply to our network model with $\mathcal{C} = 1$. The commonly cited $\Omega(D \log{(n/D)})$ bound of Kushilevitz and Mansour [25], proved for randomized distributed multihop broadcast, does *not* apply in our setting, as we assume receiver collision detection. In fact, there are no bounds, that we know of, specific to distributed broadcast with collision detection. With this in mind, we turn to the bound for *centralized* solutions to broadcast in single channel networks, from [2]. This bound proves that there exists a family of constant-diameter graphs such that every centralized broadcast algorithm requires at least $f(n) = \Omega(\log^2 n)$ rounds. Centralized solutions, of course, are stronger than randomized distributed solutions with collision detection, so a bound for the former certainly holds for the latter. By our above simulation argument, it holds that no algorithm can solve multihop broadcast in less than $f(n)/\mathcal{C} = \Omega(\frac{\log^2 n}{\mathcal{C}})$ rounds. If we replace n with $n - D$, due to our assumption that $D \leq n/2$ we get a network of size $O(n)$ that still requires $\Omega(\frac{\log^2 n}{\mathcal{C}})$ rounds to broadcast in. If we put this network on one end of a line of D nodes, and make the far end the broadcast source, the bound extends to $\Omega(D + \frac{\log^2 n}{\mathcal{C}})$.

We now continue with a lower bound for the setting with disruption ($t > 0$) and *no* common source of randomness. In Section 5, we presented a $O((D + \log n)\frac{\mathcal{C}t}{\mathcal{C}-t} \log{(\frac{n}{t})})$-time broadcast algorithm in this setting. Our lower bound below shows this to be within a factor of $O(\log{(\frac{n}{t})})$ of optimal. This bound uses the following fact, first proved in our study of the wireless synchronization problem in the t-disrupted model [12]:

Lemma 9 (Theorem 4 of [12]). *Assume there are two processes* u *and* v *attempting to communicate in a* t*-disrupted network with* $\mathcal{C}$ *channels,* $t > 0$*, and no common source of randomness. Fix a constant* ϵ*. With probability* ϵ*,* u *and* v *cannot communicate for* $\Omega(\frac{\mathcal{C}t}{\mathcal{C}-t} \log(1/\epsilon))$ *rounds.*

We use this fact to prove our bound on broadcast:

Theorem 6. *Assume no common source of randomness. It follows that every algorithm requires $\Omega((D + \log n)\frac{\mathcal{C}t}{\mathcal{C}-t})$ rounds to solve the multihop broadcast problem.*

Proof. We consider two different networks. First, consider the simple network with only two processes, u and v. Lemma 9 shows that for $\epsilon = 2/n$ there is a probability of at least $2/n$ that u and v do not communicate for $\Omega(\log n\frac{\mathcal{C}t}{\mathcal{C}-t})$ rounds.

Next, consider the "line" network consisting of a set of processes $v_0, v_2, ..., v_D$, where v_0 is the source and can communicate only with v_1, and, for $0 < i < D$, v_i can communicate only with v_{i-1} and v_{i+1}. Fix $\epsilon' = 1/4e$. By Lemma 9, we know that with probability $1-\epsilon'$, for some constant c, it takes v_i at least $c(\frac{\mathcal{C}t}{\mathcal{C}-t})$ rounds to transmit the message to v_{i+1}.

We now calculate the probability that for some $D/2$ of the v_i, the communication from v_i to v_{i+1} is faster than $c(\frac{\mathcal{C}t}{\mathcal{C}-t})$. In particular, for a given set of $D/2$ links, the probability is $\epsilon'^{D/2}$ that each communication from v_i to v_{i+1} is faster than $c(\frac{\mathcal{C}t}{\mathcal{C}-t})$. Moreover, there are at most $\binom{D}{D/2} \leq (2e)^{D/2}$ such sets of $D/2$ links. Thus, for $\epsilon' < 1/4e$, we conclude that the probability of $D/2$ links exceeding the specified speed is at most $(2e\epsilon')^{D/2} < (1/2)^{D/2} \leq 1/2$ (where $D > 1$). Thus, with probability at least $1/2$, half the links require time $\Omega(\frac{\mathcal{C}t}{\mathcal{C}-t})$, leading to a running time of $\Omega(D\frac{\mathcal{C}t}{\mathcal{C}-t})$. Combining these two claims yields the desired result.

7 Conclusion and Future Work

In this paper, we study the problem of multihop broadcast in a radio network model that assumes multiple channels and a bounded amount of adversarial disruption. We show that additional communication channels can add both *efficiency* (as compared to the single channel setting) and *robustness* (in terms of resilience to a bounded amount of adversarial communication disruption). These advantages are especially pronounced if we assume a common source of randomness. This reinforces our belief that broadcast algorithms should better leverage the multiple communication channels made available today by most popular radio protocols.

An interesting future work is to relax the assumption on the knowledge of an upper bound on t, and design algorithms that perform with running time relative to the actual amount of adversarial disruption. Another interesting future work is to design deterministic solutions that leverage the multi-selectors introduced in [17].

References

1. IEEE 802.11. Wireless LAN MAC and Physical Layer Specifications (June 1999)
2. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A Lower Bound for Radio Broadcast. Journal of Computer System Sciences 43(2), 290–298 (1991)
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. Journal of Computer and System Sciences 45(1), 104–126 (1992)

4. Bluetooth Consortium. Bluetooth Specification Version 2.1 (July 2007)
5. Chlamtac, I., Kutten, S.: On Broadcasting in Radio Networks: Problem Analysis and Protocol Design. IEEE Transactions on Communications 33(12), 1240–1246 (1985)
6. Chlamtac, I., Weinstein, O.: The wave expansion approach to braodcasting in multihop radio networks. IEEE Transactions on Communications 39, 426–433 (1991)
7. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic Broadcasting in Ad Hoc Radio Networks. Distributed Computing 15(1), 27–38 (2002)
8. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic Broadcasting in Unknown Radio Networks. In: Proceedings of the Symposium on Discrete Algorithms (2000)
9. Clementi, A., Monti, A., Silvestri, R.: Round Robin is Optimal for Fault-Tolerant Broadcasting on Wireless Networks. Journal of Parallel and Distributed Computing 64(1), 89–96 (2004)
10. Czumaj, A., Rytter, W.: Broadcasting Algorithms in Radio Networks with Unknown Topology. In: Proceedings of the Symposium on Foundations of Computer Science (2003)
11. Dolev, S., Gilbert, S., Guerraoui, R., Kowalski, D.R., Newport, C., Kohn, F., Lynch, N.: Reliable Distributed Computing on Unreliable Radio Channels. In: The Proceedings of the 2009 MobiHoc S^3 Workshop (2009)
12. Dolev, S., Gilbert, S., Guerraoui, R., Kuhn, F., Newport, C.: The Wireless Synchronization Problem. In: Proceedings of the International Symposium on Principles of Distributed Computing (2009)
13. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a Multi-Channel Radio Network: An Oblivious Approach to Coping with Malicious Interference. In: Proceedings of the International Symposium on Distributed Computing (2007)
14. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Secure Communication Over Radio Channels. In: Proceedings of the International Symposium on Principles of Distributed Computing (2008)
15. Drabkin, V., Friedman, R., Segal, M.: Efficient byzantine broadcast in wireless ad hoc networks. In: Proceedings of the Conference on Dependable Systems and Networks, pp. 160–169 (2005)
16. Gasieniec, L., Peleg, D., Xin, Q.: Faster Communication in Known Topology Radio Networks. Distributed Computing 19(4), 289–300 (2007)
17. Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C.: Interference-Resilient Information Exchange. In: The Proceedings of the Conference on Computer Communication (2009)
18. Khabbazian, M., Kowalski, D., Kuhn, F., Lynch, N.: Decomposing Broadcast Algorithms Using Abstract MAC Layers. In: Proceedings of the International Workshop on Foundations of Mobile Computing (2010)
19. Koo, C.-Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: Proceedings of the International Symposium on Principles of Distributed Computing, pp. 275–282 (2004)
20. Koo, C.-Y., Bhandari, V., Katz, J., Vaidya, N.H.: Reliable broadcast in radio networks: The bounded collision case. In: Proceedings of the International Symposium on Principles of Distributed Computing (2006)
21. Kowalski, D., Pelc, A.: Broadcasting in Undirected Ad Hoc Radio Networks. In: Proceedings of the International Symposium on Principles of Distributed Computing (2003)
22. Kowalski, D., Pelc, A.: Time of Deterministic Broadcasting in Radio Networks with Local Knowledge. SIAM Journal on Computing 33(4), 870–891 (2004)
23. Kowalski, D., Pelc, A.: Optimal Deterministic Broadcasting in Known Topology Radio Networks. Distributed Computing 19(3), 185–195 (2007)
24. Kuhn, F., Lynch, N., Newport, C.: The Abstract MAC Layer. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 48–62. Springer, Heidelberg (2009)

25. 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)
26. Newport, C.: Distributed Computation on Unreliable Radio Channels. PhD thesis. MIT (2009)
27. Pelc, A., Peleg, D.: Feasibility and Complexity of Broadcasting with Random Transmission Failures. In: Proceedings of the International Symposium on Principles of Distributed Computing (2005)
28. Richa, A., Scheideler, C., Schmid, S., Zhang, J.: A Jamming-Resistant MAC Protocol for Multi-Hop Wireless Networks. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 179–193. Springer, Heidelberg (2010)
29. Schneider, J., Wattenhofer, R.: What Is the Use of Collision Detection (in Wireless Networks)? In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 133–147. Springer, Heidelberg (2010)
30. Strasser, M., Pöpper, C., Capkun, S.: Efficient Uncoordinated FHSS Anti-jamming Communication. In: Proceedings International Symposium on Mobile Ad Hoc Networking and Computing (2009)
31. Strasser, M., Pöpper, C., Capkun, S., Cagalj, M.: Jamming-resistant Key Establishment using Uncoordinated Frequency Hopping. In: The Proceedings of the IEEE Symposium on Security and Privacy (2008)

Leader Election Using Loneliness Detection*

Mohsen Ghaffari, Nancy Lynch, and Srikanth Sastry

CSAIL, MIT
Cambridge, MA 02139, USA

Abstract. We consider the problem of leader election (LE) in single-hop radio networks with synchronized time slots for transmitting and receiving messages. We assume that the actual number n of processes is unknown, while the size u of the ID space is known, but possibly much larger. We consider two types of collision detection: *strong (SCD)*, whereby all processes detect collisions, and *weak (WCD)*, whereby only non-transmitting processes detect collisions.

We introduce *loneliness detection (LD)* as a key subproblem for solving LE in WCD systems. LD informs all processes whether the system contains exactly one process or more than one. We show that LD captures the difference in power between SCD and WCD, by providing an implementation of SCD over WCD and LD. We present two algorithms that solve deterministic and probabilistic LD in WCD systems with time costs of $\mathcal{O}(\log \frac{u}{n})$ and $\mathcal{O}(\min(\log \frac{u}{n}, \frac{\log(1/\epsilon)}{n}))$, respectively, where ϵ is the error probability. We also provide matching lower bounds.

We present two algorithms that solve deterministic and probabilistic LE in SCD systems with time costs of $\mathcal{O}(\log u)$ and $\mathcal{O}(\min(\log u, \log\log n + \log(\frac{1}{\epsilon})))$, respectively, where ϵ is the error probability. We provide matching lower bounds.

1 Introduction

We study the leader election problem in single-hop radio networks with synchronized time slots for transmitting messages, but where messages are subject to collisions. We focus on the time cost of electing a leader and the dependence of this cost on the number n of actual processes in the network as well as on a known, finite ID space I of the processes. We assume that while n may be unknown, each process knows its own ID and the ID space I. We only restrict the size of the ID space, $u = |I|$, to be at least n; the number of processes in the system may be much smaller than the size of the ID space.

The time cost of leader election depends significantly on the ability of the processes to detect message collisions. The problem has been well studied in single-hop systems which have no collision detection (*e.g.*, [3,6,10,1,8]) and in systems with *strong collision detection* (SCD) in which all processes can detect message collisions (*e.g.*, [11,2,5,3,7,9]). However, the cost of leader election is

* This work partially supported by NSF under award numbers CCF-0937274, CCF-0939370, and CCF-0726514, and by AFOSR under award number FA9550-08-1-0159.

under-explored in systems with *weak collision detection* (WCD) wherein only the listening processes detect message collisions. We focus on the time costs of solving leader election, both deterministic and randomized, in WCD systems and compare them to the time costs for leader election in SCD systems.

The primary challenge to solving leader election in WCD systems is to distinguish the following two cases: (1) $n > 1$ and all the processes transmit simultaneously resulting in a collision, which remains undetected because no process is listening, (2) $n = 1$ and any message transmitted by the process does not collide, but the successful transmission is undetected because no process is listening. In both cases the transmitting processes receive the same feedback from the WCD system despite different outcomes. Note that these two cases are distinguishable in SCD systems. Hence, *loneliness detection* — determining whether or not there is exactly one process in the system — is a key subproblem of leader election.

Summary of Results. We define the Loneliness Detection (LD) problem in Sect. 4 and determine the time complexity of solving LD in single-hop wireless networks in Sect. 5. We show that LD can be solved deterministically in WCD systems in $\mathcal{O}(\log \frac{u}{n})$ time slots for $n > 1$ and in $\mathcal{O}(\log u)$ time slots for $n = 1$. Interestingly, in the probabilistic case, if $n > 1$, LD can be solved in WCD systems in a constant number of rounds with high probability; however, if $n = 1$, then our algorithm takes $\mathcal{O}(\log u)$ time slots. We demonstrate that these time bounds are tight by presenting matching lower bounds.

In Sect. 4, we implement an SCD system on top of a WCD system augmented with a solution to LD. This allows us to deploy SCD-based protocols on WCD systems. We explore such SCD-based protocols for LE in Sect. 6 where we present upper and lower bounds for both deterministic and randomized LE in SCD systems. First, we present a deterministic LE protocol that terminates in at most $\mathcal{O}(\log u)$ time slots and show a matching lower bound. For probabilistic LE, we interleave Nakano and Olariu's algorithm from [9] with our deterministic algorithm to solve the problem in $\mathcal{O}(\min(\log u, \log\log n + \log(\frac{1}{\epsilon}))$ rounds with termination probability at least $1-\epsilon$ (where $1 < \epsilon < 0$). We present a lower bound of $\Omega(\min(\log(\frac{u}{n}), \log(\frac{1}{\epsilon})))$ for probabilistic LE on SCD systems with termination probability at least $1 - \epsilon$. Note that the lower and upper bounds match when $\epsilon = \mathcal{O}(\frac{1}{n})$. Subsequently, in Sect. 6, we demonstrate that the same upper and lower bounds hold for LE in WCD systems as well.

The full proofs omitted here due to space constraints are available in [4].

2 System Models, Definitions, and Notations

Our model considers a finite set of n processes with unique IDs from I, a finite ID space of size u. The set $J \subseteq I$ denotes the set of IDs of the n processes.

Processes. Processes communicate by broadcasting messages from a fixed alphabet $\mathcal{M}$ on the shared channel. We assume that $\mathcal{M}$ does not contain the special placeholder elements $\bot$ and $\top$, which denote silence and collision, respectively. We assume that time is divided into *rounds*, and processes have synchronized clocks and can detect the start and end of each round. Processes transmit only at round boundaries, and each transmission is contained within a single round. We assume that all processes wake up at time 0, which is the start of round 1.

A process i transmits a message m in round r through the action $send(m, r)_i$ and receives a message m' in round r through the action $receive(m', r)_i$. If a process i does not send a message in round r, then, we say that process i executes action $send(\perp, r)_i$. If a process does not send a message in round r, then it is assumed to be listening in round r. If a process i does not receive a message in round r, then the process either receives silence through action $receive(\perp, r)_i$ or receives a collision notification through action $receive(\top, r)_i$. We assume that in every execution, for each process i and each round r, exactly one event of the form $send(*, r)_i$ and exactly one event of the form $receive(*, r)_i$ occurs.

Wireless Channels. A wireless channel is a broadcast medium in which at most one process can successfully send a message in any round. We assume that a known, nondecreasing *time-bound function* $b : \mathbb{N}^+ \rightarrow \mathbb{R}^{\geq 0}$, which maps each round to an upper bound on the real time at which the round ends. Any channel that satisfies a time-bound function b is said to be a *b-time-bounded channel.*

The behavior of a channel in a round r is determined by the set T of transmitting processes in round r. If no process transmits a message in round r, *i.e.*, $|T| = 0$, then for each process i in the system, the event $receive(\perp, r)_i$ occurs; that is, all processes receive silence in round r. If exactly one process j transmits a message (say) m in round r, *i.e.*, $|T| = 1$, then for each process i in the system, the event $receive(m, r)_i$ occurs. If two or more processes send messages in a given round, *i.e.*, $|T| > 1$, then we say that the round experiences a message collision. The responses given by a channel in the event of a message collision are determined by their collision detection ability. We consider two types of channels.

Weak Collision Detection (WCD) Channels. In WCD channels, in the case of a collision, every transmitting process receives its own message, and every process that is listening receives $\top$. That is, if $|T| > 1$, then for each process i in T, where event $send(m_i, r)_i$ occurs, the event $receive(m_i, r)_i$ occurs, and for each process i not in T, the event $receive(\top, r)_i$ occurs. We denote the time-bound function of a WCD channel by b_{WCD}.

Strong Collision Detection (SCD) Channels. In SCD channels, if a message collision occurs, then all processes receive $\top$ in that round. That is, if $|T| > 1$, then for each process i, the event $receive(\top, r)_i$ occurs.

We also consider *probabilistic SCD channels* in which the real-time duration of a round is specified by a function $\rho : \mathbb{N}^+ \times [0, 1] \rightarrow \mathbb{R}^{\geq 0}$, where $\rho(r, \epsilon)$ is an upper bound on the real time by which round r terminates with probability at least $1 - \epsilon$. We assume that ρ is nondecreasing with respect to r and nonincreasing with respect to ϵ. Additionally, we assume that every round terminates in finite time with probability 1; that is, for each round r, $\rho(r, 0)$ is finite. We say that a probabilistic SCD channel whose round duration is upper bounded by a function ρ is *ρ-time-bounded.*

We assert that systems with SCD are at least as powerful as systems with WCD because SCD provides more information than WCD to the transmitting processes in a given round. A proof appears in [4].

3 The Leader Election Problem

Leader election (LE) is a problem in which each process i eventually outputs $leader(l)_i$ where l is the ID of some process in the system, and the process l is the *leader*. The safety properties of LE state that every process i performs at most one $leader_i$ event, and no two processes output different leaders.

We consider two variants of the LE problem, deterministic and probabilistic, which differ only in their liveness properties. The *deterministic liveness* property states that in any execution, for every process $i \in J$, some event of the form $leader(*)_i$ occurs. The *probabilistic liveness* property states that in the space defined by all infinite executions, for every process i some event of the form $leader(*)_i$ occurs with probability 1. For each process i, an upper bound on the number of rounds within which a $leader_i$ event occurs with probability at least $1 - \epsilon$ is given by $\rho_{LE}(\epsilon)$ where $\rho_{LE} : [0, 1] \to \mathbb{N}^+$ is a nonincreasing function. *Deterministic leader election* satisfies the safety properties and the deterministic liveness property, whereas *probabilistic leader election* satisfies the safety properties and the probabilistic liveness property.

For the purposes of demonstrating lower bounds, we also consider the variant η-LE of leader election where $\eta \in \mathbb{N}^+$. Specifically, *deterministic η-LE* denotes the variant of deterministic leader election in which the system consists of η processes and η is known to the processes; similarly, *probabilistic η-LE* denotes the variant of probabilistic leader election in which the system consists of η processes and η is known to the processes.

Prior Work. There is a significant body of work exploring LE in wireless systems with collisions. Most of the results focus on LE in SCD systems. For deterministic LE in single-hop SCD systems, there exist matching time bounds $\mathcal{O}(\log n)$ from [2,5] and $\Omega(\log n)$ from [3] where n, the number of processes in the system, is known. When n is unknown, the best known deterministic upper bound on the time complexity of LE in SCD systems is $\mathcal{O}(n)$ in [7] for arbitrary multi-hop networks of which single-hop is a special case; however, the result in [7] assumes that an upper bound u of n is known and is $\mathcal{O}(n)$. To our knowledge, better upper and lower bounds are not known.

For probabilistic LE in single-hop SCD systems, Willard presents an algorithm in [11] that solves LE on SCD systems in expected time $\mathcal{O}(1)$, $\mathcal{O}(\log \log u)$, and $\log \log n + o(\log \log n)$ in the cases where n is known, where n is unknown, but u is known, and where both n and u are unknown, respectively. For the case where n and u are unknown, the results in [9] provided an improved algorithm with running time $\log \log n + o(\log \log n) + \mathcal{O}(\log(\frac{1}{\epsilon}))$ with probability of termination at least $1 - \epsilon$. A lower bound of $\Omega(\log \frac{1}{\epsilon})$ for this problem has been presented in [9] only for "uniform algorithms" in which all the processes transmit with the same probability in each round (although the probability can vary from one round to another).

To our knowledge, the problem of LE seems to be relatively under-explored in the context of WCD systems. The best known time bounds for deterministic LE in single-hop WCD systems, based on the results for broadcast in multi-hop wireless networks in [10], is $\Theta(\log n)$ where n is known.

4 The Loneliness Detection Problem

Loneliness detection (LD) is a service that interacts with processes through output *alone*. In some round r, LD outputs $alone(a, r)_i$ for all processes i where a is Boolean. The safety properties state that if a is *true* there is exactly one process in the system, and if a is false, then there is more than one process. Note that LD outputs its *alone* event at all processes in the same round.

We consider two variants of LD, deterministic and probabilistic, which differ only in their liveness properties. The *deterministic liveness* property states that in any execution, for each process i, some $alone_i$ event occurs. The *probabilistic liveness* property states that in the probability space defined by all infinite executions, for each process i, some $alone_i$ event occurs with probability 1. The upper bound on the number of rounds within which some *alone* event occurs with probability at least $1-\epsilon$ is given by $\rho_{LD}(\epsilon)$ where $\rho_{LD} : [0,1] \to \mathbb{N}^+$ is non-increasing, and $\rho_{LD}(0)$ is finite. A *deterministic loneliness detector* satisfies the safety and the deterministic liveness properties, whereas a *probabilistic loneliness detector* satisfies the safety and the probabilistic liveness properties.

A solution to deterministic LD in which an *alone* event occurs within r_{LD} rounds is said to be a r_{LD}*-round-bounded*, and a solution to randomized LD in which some *alone* event occurs with probability at least $1-\epsilon$ within $\rho_{LD}(\epsilon)$ rounds is said to be $\rho_{LD}(\epsilon)$*-round-bounded.*

We show that Loneliness Detection (LD) is, in a sense, exactly the difference between SCD and WCD. Note that LD is solved on SCD systems using the following trivial algorithm. Each process i in the system sends a message m at the beginning of round 1 and waits until the end of round 1. If a $\top$ is received at the end of round 1, then the algorithm returns $alone(false, 1)_i$, otherwise it returns $alone(true, 1)_i$.

We now present an algorithm that implements an SCD channel over a WCD system augmented with an LD service. In order to distinguish the actions of the two channels, we rename the *send* and *receive* actions associated with the WCD channel as $sendWCD$ and $receiveWCD$ actions, respectively.

Pseudocode Notation. When an action at a process is triggered by an event e, we denote the trigger with "**upon** e" in the pseudocode. Similarly, when the automaton is waiting for the occurrence of an event e to proceed, we denote it with "**wait until** e". Instances in which an algorithm performs an action a are denoted "**perform** a".

We also use the following notation to bind values to certain variables. Consider the statements "upon $e(x, y)$" and "wait until $e(x, y)$". In both cases, if (say) x is undefined and y is defined at the point in the code where the statements occur, then the semantics of the code is to wait for any event of the form $e(*, y)$, and when an event (say) $e(x', y)$ occurs, bind the value of x' to x.

Algorithm Description. The algorithm consists of two concurrent tasks: Init and Communicate. In the Init task each process i waits for the $alone(a_{LD}, r_{LD})_i$ event from the LD service. The Communicate task consists of two WCD rounds, called Transmit and Ack, which are executed for every round r_s of the SCD channel. Let T denote the set of processes that transmit some message m_i in

Algorithm 1. Implementing SCD on a WCD system using an LD service

```
Each process $i$ executes two concurrent tasks: Init and Communicate.
Variables:
    $m \in \mathcal{M} \cup \{\perp\}$
    $m', p2msg \in \mathcal{M} \cup \{\top, \perp\}$
    $r_{LD}, r_s \in \mathbb{N}^+$
**Task Init:**
    Wait until event $alone(a_{LD}, r_{LD})_i$ from the LD service; **halt**
**Task Communicate:**
    loop forever
        /* Round $r_s$ for the SCD channel starts here */
        upon $send(m, r_s)_i$ wait for Task Init to terminate
        **Transmit Round:**
            perform $sendWCD(m, r_{LD} + 2r_s - 1)_i$
            wait until $receiveWCD(m', r_{LD} + 2r_s - 1)_i$
        **Ack Round:**
            if ($m = \perp$ and $m' \in \mathcal{M}$) then perform $sendWCD(\text{“}ack\text{”}, r_{LD} + 2r_s)_i$
            else perform $sendWCD(\perp, r_{LD} + 2r_s)_i$
            wait until $receiveWCD(p2msg, r_{LD} + 2r_s)_i$
        if ($m' = \perp$) then perform $receive(\perp, r_s)_i$
        else if ($a_{LD} = true$) then perform $receive(m, r_s)_i$
        else if ($p2msg \neq \perp$) then perform $receive(m', r_s)_i$
        else perform $receive(\top, r_s)_i$
        /* Round $r_s$ ends here */
    end loop
```

round r_s of the SCD channel via $send(m_i, r_s)_i$ for each process $i \in T$. In the Transmit round, each process $i \in T$ executes $sendWCD(m_i, r_{LD} + 2r_s - 1)_i$. All other processes listen to the channel via $sendWCD(\perp, r_{LD} + 2r_s - 1)_i$. At the end of the Transmit round, each process $i \in T$ receives its own message via the event $receiveWCD(m', r_{LD} + 2r_s - 1)_i$ where $m' = m_i$. Each listening process j receives either some message, $\perp$, or $\top$ via $receiveWCD(m', r_{LD} + 2r_s - 1)_j$. In the Ack round, each process $i \in T$ listens to the channel via the event $sendWCD(\perp, r_{LD} + 2r_s)_i$. Each process $j \in J \setminus T$ sends an “*ack*” via the event $sendWCD(\text{“}ack\text{”}, r_{LD} + 2r_s)_j$ iff j received a message in the Transmit round; otherwise j listens to the channel via the event $sendWCD(\perp, r_{LD} + 2r_s)_j$. At the end of the Ack round, each process receives either “*ack*”, $\perp$, or $\top$ via $receiveWCD(p2msg, r_{LD} + 2r_s)_*$. If $p2msg$ is “*ack*” or $\top$, then the transmission in the Transmit round was successful, and m' is that transmitted message; so the algorithm outputs $receive(m', r_s)_i$ at each process i. If a_{LD} is $true$, then there is only one process in the system, and so the transmission in the Transmit round was successful, and the algorithm outputs $receive(m, r_s)_i$ at the lone process i. However, if a_{LD} is $false$ and $p2msg$ is $\perp$, then there was a collision, and the algorithm outputs $receive(\top, r_s)_i$ at each process.

Note that although the Init and Communicate tasks are executed concurrently, the Communicate task waits for the Init task to terminate before proceeding to sending and receiving messages on the WCD channel. The pseudocode is shown in Algorithm 1.

Theorem 1. *Algorithm 1 implements a deterministic SCD channel over a WCD system with a deterministic LD service. If the WCD channel is b_{WCD}-time-bounded and the LD service is r_{LD}-round-bounded, then the SCD channel implementation is b-time-bounded, where $b(r) = b_{WCD}(r_{LD} + 2r)$.*

Proof Sketch. We show that (1) if no process sends a message in a round (say) r_s, then all the processes receive $\perp$ in round r_s; (2) if exactly one process sends a message (say) m in round r_s, then all the processes receive m in the round r_s; and (3) if multiple processes send messages in round r_s, then all the processes receive $\top$ in round r_s.

Properties (1) and (2) are easy to verify based on the following observations. When $n = 1$, at the lone process i, the event $alone(true, r_{LD})_i$ occurs, and when $n > 1$, at each process i, the event $alone(false, r_{LD})_i$ occurs. For property (3) let T denote the set of transmitting processes in round r_s. If $|T| \geq 2$, then there is a message collision. From the properties of a WCD channel, we know that for every process $j \in T$, m'_j is m_j, and for every process $i \notin T$, m'_i is $\top$ (and m_i is $\perp$). Therefore, in the Ack round, for each process i event $sendWCD(\perp, r_{LD} + 2r_s)$ occurs, and consequently, $p2msg_i$ is $\perp$. Given that $a_{LD} = false$, and $m'_i \neq \perp$ at every process i, each process i executes $receive(\top, r_s)_i$. That is, if $|T| \geq 2$, then all processes receive $\top$ in round r_s.

Next we determine a time-bound function b for the SCD channel. Let b_{WCD} denote a time-bound function for the underlying WCD channel. From the pseudocode we know that the deterministic LD service outputs the *alone* events in round r_{LD} of the WCD channel, and subsequently between every pair of events $send(*, r)_i$ and $receive(*, r)_i$, there are exactly two *sendWCD* events of the form $sendWCD(*, r_{LD} + 2r - 1)_i$ and $sendWCD(*, r_{LD} + 2r)_i$. Therefore, $b(1) = b_{WCD}(r_{LD} + 2)$ and for each round r, $b(r) = b_{WCD}(r_{LD} + 2r)$. □

Additionally, if the underlying LD service is a $\rho_{LD}(\epsilon)$-round-bounded probabilistic LD service, then Algorithm 1 implements a $\rho(r, \epsilon)$-time-bounded probabilistic SCD channel where $\rho(r, \epsilon) = b_{WCD}(\rho_{LD}(\epsilon) + 2r)$. Thus, we have the following result whose correctness proof is similar to Theorem 1.

Theorem 2. *Algorithm 1 implements probabilistic SCD channel over a WCD channel and a probabilistic LD service. If the WCD channel is b_{WCD}-time-bounded and the probabilistic LD service is $\rho_{LD}(\epsilon)$-round-bounded, then the probabilistic SCD channel implementation is $\rho(r, \epsilon)$-time-bounded where $\rho(r, \epsilon) = b_{WCD}(\rho_{LD}(\epsilon) + 2r)$.*

Remark 1. Implementing LD on top of a WCD system augmented with a solution to LE takes just one additional round. First, all the processes elect a leader with the assumed solution to LE. In the next round of the assumed WCD system, (1) every process that is not the leader transmits, outputs $false$, and then halts; (2) concurrently, the leader outputs $true$ iff it receives $\perp$ at the end of this round, and outputs $false$ otherwise. Correctness is straightforward.

5 Algorithms and Lower Bounds for Loneliness Detection

Here, we explore algorithms and lower bounds for Loneliness Detection. Since LD can be solved in SCD systems in a single round, we focus on WCD systems.

5.1 Upper Bounds for LD in WCD Systems

We present two algorithms, one deterministic and the other randomized. The deterministic algorithm solves the deterministic LD problem in $\mathcal{O}(\log(\frac{u}{n-1}))$ rounds whereas the randomized algorithm solves the probabilistic LD problem in $\mathcal{O}(\frac{\log(1/\epsilon)}{n-1})$ rounds with probability $1-\epsilon$, for $\epsilon \in (0,1]$. We also combine these algorithms to solve probabilistic LD with probability 1.

Bitwise Separation Protocol (BSP). BSP solves the deterministic LD problem in WCD systems in $\mathcal{O}(\log \frac{u}{n-1})$ rounds. The algorithm is as follows. Let the ID of each process i be represented as a sequence of bits denoted id_i; since the ID space is of size u, the sequence is $\lceil \log(u) \rceil$ bits long. Starting from the least significant bit, number the bits from 1 to $\lceil \log(u) \rceil$, and let $id_i[k]$ denote the k-th bit of process i's ID. Let $T_k = \{i \in J : id_i[k] = 1\}$[1]. The algorithm proceeds in phases, each phase consisting of a Transmit round and an Ack round.

In the Transmit round of the k-th phase, exactly the processes in T_k that have not yet halted transmit a message. In the Ack round, if a process $i \in J \setminus T_k$ that has not halted receives either a message or $\top$ in the Transmit round, then i sends an "*ack*" message; furthermore, processes in T_k do not send a message in the Ack round. If a process $i \in J$ that has not yet halted either sends or receives an "*ack*" message or receives $\top$ in the Ack round of a given phase k, then i outputs $alone(false, 2k)_i$ and halts.

If $\perp$ is received in all the Ack rounds, then the algorithm terminates at the end of $2\lceil \log u \rceil$ rounds and the lone process (say) j outputs $alone(true, 2\lceil \log u \rceil)_j$. The pseudocode is shown in Algorithm 2.

Theorem 3. *BSP solves the deterministic LD problem and for each process i; some* $alone_i$ *event occurs after* $2\lceil \log(u) \rceil$ *rounds if* $n = 1$ *and within* $2(\lceil \log u \rceil - \lceil \log n \rceil + 1)$ *rounds if* $n > 1$.

Proof Sketch. From the pseudocode in Algorithm 2, note that every process i performs exactly one $alone_i$ event. First, assume that $alone(true, r)_i$ occurs. From the pseudocode, we see that $r = 2\lceil \log u \rceil$. For the purpose of contradiction, we assume that $n > 1$. Since no $alone(false, *)_i$ event occurs, i never sends an "*ack*" and i never receives an "*ack*" or $\top$ in any Ack round. This can happen only if in the Transmit round of every phase $k \leq \lceil \log u \rceil$, either all the processes transmit or all the processes listen; this implies that all the processes share the same ID. This contradicts our assumption that process IDs are unique. Hence, when $n = 1$, event $alone(true, 2\lceil \log u \rceil)_i$ occurs at the lone process i.

Alternatively, assume that $alone(false, r)_i$ occurs. From the pseudocode, we know that $m_i'' \neq \perp$ in round r; that is, i either sent or received an "*ack*" message

[1] Recall that J is the set of processes comprising the system.

Algorithm 2. Bitwise Separation Protocol

```
Let $m \in \mathcal{M}$ denote a message that $i$ sends to signal its presence in the system.
Process $i$ executes the following:
    for $k := 1$ to $\lceil \log(u) \rceil$
        **Transmit round:**
            if $(id[k] = 1)$ then perform $send(m, 2k-1)_i$
            else perform $send(\perp, 2k-1)_i$
            wait until $receive(m', 2k-1)_i$
        **Ack round:**
            if $(id[k] \neq 1$ and $m' \neq \perp)$ then perform $send(\text{“}ack\text{”}, 2k)_i$
            else perform $send(\perp, 2k)_i$
            wait until $receive(m'', 2k)_i$
            if $(m'' \neq \perp)$ then perform $alone(false, 2k)_i$; **halt.**
    endfor
    perform $alone(true, 2\lceil \log(u) \rceil)_i$
```

or received $\top$ in the Ack round of phase $r/2$. Therefore, there is at least one other process in the system. Furthermore, since i either sent or received an "*ack*" message or received $\top$ in round r, then the properties of the WCD channel imply that the same is true for all processes. Hence, for each process j, the event $alone(false, r)_j$ occurs.

Now we provide an upper bound on r. Since representing unique IDs among n processes requires $\lceil \log n \rceil$ bits, and since each process ID is of length $\lceil \log u \rceil$ bits, we infer that within the first $\lceil \log u \rceil - \lceil \log n \rceil + 1$ bits of the process IDs, at some bit position k, there exist at least two processes i and j such that $id_i[k] = 1$ and $id_j[k] = 0$. Therefore, in phase k, i transmits and j listens in the Transmit round. Hence, $alone(false, 2k)$ event occurs at each process. □

Random Separation Protocol (RSP). RSP is used to solve probabilistic LD in WCD systems. RSP verifies that $n > 1$ in $2\frac{log(1/\epsilon)}{n-1}$ rounds with probability at least $1 - \epsilon$ where $\epsilon \in (0, 1]$. However, if $n = 1$, RSP does not terminate.

The protocol is identical to BSP except that IDs in RSP are infinite-bit strings in which the bits is chosen independently and uniformly at random. In each phase k, if $id_i[k] = 1$, then i transmits in the Transmit round; otherwise i listens in the Transmit round. It can be verified easily that RSP terminates in the first phase k in which for some pair of processes i and j, $id_i[k] \neq id_j[k]$.

The probability that the k-th bit of the IDs of all processes are identical is $2 \cdot (\frac{1}{2})^n = (\frac{1}{2})^{n-1}$. It follows that, for a given $\epsilon \in (0, 1]$, the probability that RSP does not terminate in $\frac{log(1/\epsilon)}{n-1}$ phases is ϵ.

Theorem 4. *In RSP, if $n > 1$ and $\epsilon \in (0, 1]$, then for every process i the event $alone(false, r)_i$ occurs within the first $\frac{log(1/\epsilon)}{n-1}$ phases, that is, $r \leq 2\frac{log(1/\epsilon)}{n-1}$, with probability at least $1 - \epsilon$.*

Remark 2. For $\epsilon = 2^{-n}$, Theorem 4 implies that, if $n > 1$, then for every process i, the event $alone(false, r)_i$ occurs within the first $2\frac{n}{n-1}$ rounds with probability at least $1 - 2^{-n}$. Thus, for $n > 1$, the number of rounds within which RSP terminates with high probability is always at most 4.

Combined Separation Protocol (CSP). Even though RSP terminates in fewer rounds than BSP with high probability if $n > 1$, it fails to terminate if $n = 1$. We overcome this problem by interleaving RSP and BSP, executing BSP in the odd rounds and RSP in the even rounds; CSP terminates when either BSP or RSP terminates.

Theorem 5. *CSP solves probabilistic LD on WCD systems where*

$$\rho_{LD}(\epsilon) = \begin{cases} 4\lceil \log u \rceil & \textit{if } n = 1, \\ 4(\lceil \log u \rceil - \lceil \log n \rceil + 1) & \textit{if } \epsilon = 0 \textit{ and } n > 1, \\ 4 \min\left((\lceil \log u \rceil - \lceil \log n \rceil + 1), \frac{\log(1/\epsilon)}{n-1} \right) & \textit{if } \epsilon \in (0, 1] \textit{ and } n > 1. \end{cases}$$

5.2 Lower Bounds for LD in WCD Systems

In this section, we present lower bounds for both probabilistic and deterministic LD problems in WCD systems.

Lemma 1. *For any LD algorithm A for WCD systems, any $n > 1$, and any round number $r \leq \lceil \log(u) \rceil - \lfloor \log(n-1) \rfloor - 2$, there exists a set J_{LB} of n processes, such that, when A is run on the system consisting of all processes in J_{LB}, the probability that A does not terminate within r rounds, is at least $(\frac{1}{2})^{rn}$.*

Proof Sketch. For each process i, we consider executing A on system S_i consisting of only process i. Without loss of generality, assume that each execution of S_i takes at least r rounds. Consider the probability space of all executions corresponding to r rounds of S_i. For each such execution and each round z, $1 \leq z \leq r$, define $trans_i(z)$ to be $true$ if i transmits in round z, and $false$ otherwise. Denote the probability of events in this space by Pr_i. We define the boolean function dtd_i (*dominating transmission decision*) on $\{1, ..., r\}$ recursively. $dtd_i(1)$ is assigned the value that is more likely to be taken by $trans_i(1)$, *i.e.*, $dtd_i(1) = true$ iff $Pr_i\{trans_i(1) = true\} \geq \frac{1}{2}$. Let $DTD_{i,1}$ denote the event in the probability space Pr_i that $trans_i(1) = dtd_i(1)$. For each $z \geq 2$, we define $dtd_i(z)$ to be the value that is more likely to be taken by $trans_i(z)$, conditioned on $DTD_{i,z-1}$, *i.e.*, $dtd_i(z) = true$ iff $Pr_i\{trans_i(z) = true | DTD_{i,z-1}\} \geq \frac{1}{2}$. We denote by $DTD_{i,z}$ the event that for each round r', $1 \leq r' \leq z$, $trans_i(r') = dtd_i(r')$.

Since for each process i and each round z, $1 \leq z \leq r$, there are two possible values for $dtd_i(z)$, there are 2^r possible values for dtd_i sequences. Since $2^r < \frac{u}{n-1}$, by the Pigeonhole principle, there exists a set J_{LB} of n processes that have identical sequences of dominating transmission decisions, *i.e.*, $\forall i, j \in J_{LB}, dtd_i = dtd_j$. Let S be the system consisting of processes in J_{LB}. For each z, $1 \leq z \leq r$, let $cdtd(z)$ denote the common dominating transmission decision of the processes of J_{LB} in round z.

Consider an execution α in system S. If for each process $i \in J_{LB}$ and each round $z \leq r$ of α, $trans_i(z) = cdtd(z)$, then for each $i \in J_{LB}$ there exists an execution β in S_i such that i cannot distinguish α from β in the first r rounds. However, in α, the only valid output is $alone(false, *)_*$ whereas in β, the only valid output is $alone(true, *)_*$. Hence, j cannot terminate within r rounds. By induction we show that the probability that for each $i \in J_{LB}$ and z, $1 \leq z \leq r$, $trans_i(z) = cdtd(z)$ is at least $(\frac{1}{2})^{rn}$. Hence, with probability at least $(\frac{1}{2})^{rn}$, A does not terminate within r rounds. □

Lemma 2. *For $n = 1$, no LD algorithm for WCD systems guarantees termination within $\lceil \log(u) \rceil - 2$ rounds.*

Theorem 6. *For any LD algorithm A for WCD systems, and any $n > 1$, there exists a set of processes, J_{LB}, where $|J_{LB}| = n$, such that the probability that A terminates within $\min\left(\frac{\log(\frac{1}{\epsilon})}{n}, \lceil \log \frac{u}{n-1} \rceil - 2\right)$ rounds, when run on the system consisting of all processes of J_{LB}, is at most $1 - \epsilon$. For $n = 1$, no LD algorithm for WCD systems guarantees termination within $\lceil \log(u) \rceil - 2$ rounds.*

Proof. For $n > 1$, the proof follows by substituting $r = \min\left(\frac{\log(\frac{1}{\epsilon})}{n}, \lceil \log \frac{u}{n-1} \rceil - 2\right)$ in Lemma 1. For $n = 1$, the proof follows from Lemma 2. □

Theorem 7. *For $n > 1$, no deterministic LD algorithm for WCD systems guarantees termination within $\lceil \log \frac{u}{n-1} \rceil$ rounds. For $n = 1$, no deterministic LD algorithm guarantees termination within $\lceil \log(u) \rceil - 2$ rounds.*

5.3 Revisiting SCD on WCD Systems

In Sect. 4, we presented an implementation of an SCD channel over a WCD channel using an LD service. In Sect. 5.1, we presented the BSP and CSP algorithms that solve deterministic and probabilistic LD, respectively, over WCD. Note that, BSP and CSP do not send any messages on the WCD channel after they terminate. Hence, Algorithm 1 may use the WCD channel after round r in isolation. Therefore, BSP and CSP can be used as an LD service in Algorithm 1 to implement deterministic and probabilistic SCD systems, respectively.

6 Algorithms and Lower Bounds for Leader Election

In this section, we study deterministic and probabilistic LE problems in both SCD and WCD systems and demonstrate matching upper and lower bounds.

6.1 Upper Bounds for LE in SCD Systems

In this section, we present two algorithms: Bitwise Leader Election Protocol (BLEP) and Combined Leader Election Protocol (CLEP). The former is a deterministic algorithm which solves deterministic LE in SCD systems. The latter is a randomized algorithm which interleaves BLEP and Nakano and Olariu's algorithm in [9] to solve probabilistic LE in SCD systems.

Algorithm 3. Bitwise Leader Election Protocol

```
$active = true$
for $r := 1$ to $\lfloor \log(u) \rfloor + 1$
    if ($active = true$ and $id_i[r] = 1$) then perform $send(id_i, r)_i$
    else perform $send(\perp, r-1)_i$
    wait until $receive(m', r-1)_i$
    if ($m' \in I$) then perform $leader(m')_i$; **halt.**
    if ($m' = \top$) then $active = (active)\&(id_i[r] = 1)$
endfor
```

Bitwise Leader Election Protocol (BLEP). BLEP solves deterministic LE in SCD systems in $\mathcal{O}(\log u)$ rounds. In BLEP, every process contending to be the leader is *active*, and *inactive* otherwise; a Boolean variable *active* denotes whether or not a process is active. Initially, all the processes are *active*. The execution proceeds from round 1 to round $\lfloor \log u \rfloor + 1$. In each round r, every process i such that i is *active* and $id_i[r] = 1$, transmits its ID id_i; all other processes are silent in round r. At the end of round r, every process j receives some response from the SCD channel. If j receives a collision notification $\top$ at the end of round r, and j was *active* but did not transmit its ID in round r (because $id_j[r] = 0$), then j ceases to be *active* (becomes inactive) at the end of round r and therefore stops contending to be the leader. On the other hand, if the response at the end of round r is the ID of some process l, then j elects l as the leader, outputs $leader(l)_j$, and halts. If j receives $\perp$ at the end of round r, then j does nothing. The execution proceeds to round $r+1$, and so on, until round $\lfloor \log u \rfloor + 1$. The pseudocode is shown in Algorithm 3.

Theorem 8. *BLEP solves the LE problem within* $\lfloor \log(u) \rfloor + 1$ *rounds.*

Proof Sketch. Establishing the safety properties of LE is straightforward and follows from the pseudocode. Next, we prove that some *leader* event occurs within $\lfloor \log u \rfloor + 1$ rounds.

From the pseudocode, we see that if i and j are active in round r, then for each r', $1 \leq r' \leq r$, $id_i[r'] = id_j[r']$. Therefore, at the end of $\lceil \log u \rceil$ rounds, there can be at most two active processes in the system. If just one process (say) i remains active, then consider the earliest round $r \leq \lfloor \log u \rfloor$ at the end of which i is the only active process. By construction, multiple processes are active at the beginning of round r. Since i is the only process active at the end of round r, for each process $j \neq i$ that is active in round r, $id_j[r] = 0$ and $id_i[r] = 1$. Hence, in round r, only i transmits its ID, and all the processes in the system receive i's ID. Therefore, for each process j in the system, $leader_j$ event occurs in round $r \leq \lfloor \log u \rfloor$ and contradicts our assumption that no *leader* event occurs by the end of round $\lfloor \log u \rfloor$.

On the other hand, if two processes (say) i and j remain active, then the first $\lfloor \log u \rfloor$ bits of their IDs are identical. Therefore, the last bit of their IDs must be different. Without loss of generality, let $id_i[\lfloor \log u \rfloor + 1] = 1$ and $id_j[\lfloor \log u \rfloor + 1] =$

0. In round $\lfloor \log u \rfloor + 1$ only i transmits its ID and therefore, for each process i, $leader_i$ event occurs in round $\lfloor \log u \rfloor + 1$. □

Combined Leader Election Protocol (CLEP). Nakano and Olariu [9] present a randomized LE algorithm for SCD systems that terminates in $\mathcal{O}(1)$ rounds if $n = 1$, and in $\mathcal{O}(\log\log n + \log(\frac{1}{\epsilon}))$ rounds with probability at least $1 - \epsilon$ if $n > 1$. CLEP interleaves Nakano-Olariu's algorithm with BLEP, by executing BLEP in the odd rounds and Nakano-Olariu in the even rounds. The time complexity of CLEP matches the lower bound for probabilistic LE.

Theorem 9. *CLEP solves the LE problem in SCD systems and terminates within $\rho_{LE}(\epsilon)$ rounds with probability at least $1 - \epsilon$ where:*

$$\rho_{LE}(\epsilon) = \begin{cases} \mathcal{O}(1) & \textit{if } n = 1, \\ \mathcal{O}(\log u) & \textit{if } \epsilon = 0 \textit{ and } n > 1, \\ \mathcal{O}(\min(\log u, \log\log n + \log(\frac{1}{\epsilon})) & \textit{if } \epsilon \in (0,1] \textit{ and } n > 1. \end{cases}$$

6.2 Lower Bounds for LE in Both SCD and WCD Systems

Here we present lower bounds for deterministic and probabilistic LE in SCD systems in Theorems 10 and 12, respectively, and they match the upper bounds presented in Theorems 8 and 9, respectively. Note that these lower bounds hold for the weaker WCD systems as well. We demonstrate these lower bounds by proving the lower bounds for η-LE, and since η-LE is equivalent the LE problem where $n = \eta$ is known, lower bounds for η-LE hold for LE as well.

Lemma 3. *For any $\eta > 1$ and any randomized η-LE algorithm A in SCD systems, there exist some set J of η processes such that there is a non-zero probability that A, when run on J, does not terminate within $\lceil \log \frac{u}{\eta - 1} \rceil - 2$ rounds.*

Proof Sketch. Let $k = \lceil \log \frac{u}{\eta - 1} \rceil - 2$. Assume for contradiction that there exists an $\eta > 1$ and an η-LE algorithm A that terminates within k rounds with probability 1. We construct executions of A for each process i in which i receives $\top$ from the channel in each round that it transmitted and receives $\perp$ otherwise. Using techniques from Lemma 1, we show that there exists an execution of A on some set J_{LB} of η processes for which either each process $i \in J_{LB}$ elects itself as the leader or no process is elected leader within k rounds. This violates the properties of η-LE and forces the contradiction. □

Theorem 10. *For any $n > 1$, no deterministic LE algorithm in SCD systems can guarantee termination within $\lceil \log \frac{u}{n-1} \rceil - 2$ rounds.*

The proof follows from Lemma 3. Next we derive a lower bound for termination probability of 2-LE and extend it to the LE problem.

Lemma 4. *Let A be any 2-LE algorithm in SCD systems and suppose that $r < \lceil \log(u) \rceil - 1$ and $2 \leq u$. There exist two processes such that the probability of termination of A within k rounds, when A is run on the system of those two processes, is at most $1 - (\frac{1}{4})^{r+1}$.*

Proof Sketch. The proof structure is similar to that of Lemma 1. The key difference is the following. In the proof of Lemma 1 we considered some specific executions of an LD algorithm A_{LD} in WCD systems with just one process and showed that such executions are locally indistinguishable from some (other) specific executions of A_{LD} in a WCD system with a specific set of n processes. In SCD systems, such a construction is not feasible for the following reason. In WCD systems when a transmitting process always receives the same feedback from the channel. On the other hand, in SCD systems, a transmitting process could receive different feedback depending on whether or not a collision occurred. To circumvent this issue, we consider executions of A, a solution to 2-LE problem in SCD systems, in a fake scenario where a process receives $\top$ in every round that it transmits. We use such executions to demonstrate that with probability at least $(\frac{1}{4})^{r+1}$, A does not terminate. □

Theorem 11. *For any $u \geq 2$, any ID space I of size u, any $\epsilon \in (0, 1]$, and any 2-LE algorithm A in SCD systems, there exist two processes with IDs from I such that, when A is run with just those two processes, the probability that A terminates within $\min(\log(\frac{1}{4\epsilon})/2, \lceil\log(u)\rceil - 2)$ rounds is at most $1 - \epsilon$.*

Theorem 12. *For any $u \geq 2$, any ID space I of size u, any $\epsilon \in (0, 1]$, any η, $1 \leq \eta \leq \frac{u}{2}$, and any 2η-LE algorithm A in SCD systems, there exist 2η processes with IDs from I such that, when A is run with just those 2η processes, the probability that A terminates within $\min(\log(\frac{1}{4\epsilon})/2, \lceil\log(\frac{u}{n})\rceil - 2)$ rounds is at most $1 - \epsilon$.*

Proof Sketch. Assume for the sake of contradiction that there exists some 2η-LE algorithm A that terminates within $\min(\log(\frac{1}{4\epsilon})/2, \lceil\log(\frac{u}{n})\rceil - 2)$ rounds with probability greater than $1 - \epsilon$. Consider an ID space I^* of size $u^* = \lfloor\frac{u}{\eta}\rfloor$. Using A we construct a 2-LE algorithm A^* that emulates A on groups of processes and terminates when A does. Since A terminates within $\min(\log(\frac{1}{4\epsilon})/2, \lceil\log u^*\rceil - 2)$ rounds with probability greater than $1 - \epsilon$, the same bounds apply to A^* as well, and this contradicts contradicts Theorem 11. □

6.3 Leader Election in Weak Collision Detection Systems

In this section, we show that the LE problem in WCD systems can be solved in time complexities that match the lower bounds presented in Sect. 6.2 for both deterministic and probabilistic cases. We can solve LE on WCD systems by first implementing SCD systems on WCD systems as presented in Sect. 5.3, and then solving LE on the thus constructed SCD systems using BLEP and CLEP from Sect. 6.1. Thus, we have the following results.

Theorem 13. *For a WCD system with IDs from an ID-space of size u and consisting of n processes, $1 \leq n \leq u$, there exists a deterministic LE algorithm with a time-bound function given by $r_{LE} = \mathcal{O}(\log u)$ rounds.*

Note that the time complexity above, $\mathcal{O}(\log u)$, matches the $\Omega(\log\frac{u}{n})$ lower bound presented in Lemma 3 asymptotically when $n << u$.

Theorem 14. *For a WCD system with IDs from an ID-space of size u and consisting of n processes, $1 \le n \le u$, there exists a randomized LE algorithm with a time-bound function $\rho_{LE}(\epsilon)$ where for any $\epsilon \in (0, 1]$,*

$$\rho_{LE}(\epsilon) = \begin{cases} b_{WCD}(\mathcal{O}(\log(u))) & \textit{if } n = 1 \\ b_{WCD}(\mathcal{O}(\min(\log u, \log\log n + \log(\frac{1}{\epsilon})))) & \textit{if } n > 1. \end{cases}$$

The upper bounds presented above match the respective lower bounds. For $n = 1$, Theorem 6, along with the reduction of LD to LE in the Remark 1, shows an $\Omega(\log u)$ lower bound for LE in WCD systems which matches the upper bound. For $n > 1$, the upper bound presented above matches the lower bound presented in Theorem 12, when $\epsilon = \mathcal{O}(\frac{1}{n})$.

References

1. Bordim, J.L., Ito, Y., Nakano, K.: Randomized leader election protocols in noisy radio networks with a single transceiver. In: Guo, M., Yang, L.T., Di Martino, B., Zima, H.P., Dongarra, J., Tang, F. (eds.) ISPA 2006. LNCS, vol. 4330, pp. 246–256. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11946441_26
2. Capetanakis, J.I.: Tree algorithms for packet broadcast channels. IEEE transactions on information theory 25(5), 505–515 (1979), http://dx.doi.org/10.1109/TIT.1979.1056093
3. Clementi, A.E.F., Monti, A., Silvestri, R.: Distributed broadcast in radio networks with unknown topology. Theoretical Computer Science 302, 337–364 (2003), http://dx.doi.org/10.1016/S0304-3975(02)00851-4
4. Ghaffari, M., Lynch, N., Sastry, S.: Leader election using loneliness detection. Tech. Rep. MIT-CSAIL-TR-2011-xxx, CSAIL. MIT (2011)
5. Hayes, J.: An adaptive technique for local distribution. IEEE Transactions on Communication 26, 1178–1186 (1978), http://dx.doi.org/10.1109/TCOM.1978.1094204
6. Kowalski, D., Pelc, A.: Broadcasting in undirected ad hoc radio networks. Distributed Computing 18(1) (2005), http://dx.doi.org/10.1007/s00446-005-0216-7
7. 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), http://dx.doi.org/10.1007/978-3-642-02930-1_43
8. Nakano, K., Olariu, S.: Randomized leader election protocols in radio networks with no collision detection. In: Lee, D.T., Teng, S.-H. (eds.) ISAAC 2000. LNCS, vol. 1969, pp. 362–373. Springer, Heidelberg (2000), http://dx.doi.org/10.1007/3-540-40996-3_31
9. Nakano, K., Olariu, S.: Uniform leader election protocols for radio networks. IEEE transactions on parallel and distributed systems 13(5) (2002), http://dx.doi.org/10.1109/TPDS.2002.1003864
10. Schneider, J., Wattenhofer, R.: What is the use of collision detection (in wireless networks)? In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 133–147. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15763-9_14
11. Willard, D.E.: Log-logarithmic selection resolution protocols in a multiple access channel. SIAM Journal of Computing 15, 468–477 (1986), http://dx.doi.org/10.1137/0215032

Optimal Random Sampling from Distributed Streams Revisited

Srikanta Tirthapura[1] and David P. Woodruff[2]

[1] Dept. of ECE, Iowa State University, Ames, IA, 50011, USA
snt@iastate.edu

[2] IBM Almaden Research Center, San Jose, CA, 95120, USA
dpwoodru@us.ibm.com

Abstract. We give an improved algorithm for drawing a random sample from a large data stream when the input elements are distributed across multiple sites which communicate via a central coordinator. At any point in time the set of elements held by the coordinator represent a uniform random sample from the set of all the elements observed so far. When compared with prior work, our algorithms asymptotically improve the total number of messages sent in the system as well as the computation required of the coordinator. We also present a matching lower bound, showing that our protocol sends the optimal number of messages up to a constant factor with large probability. As a byproduct, we obtain an improved algorithm for finding the heavy hitters across multiple distributed sites.

Keywords: distributed streams, sampling, reservoir sampling.

1 Introduction

For many data analysis tasks, it is impractical to collect all the data at a single site and process it in a centralized manner. For example, data arrives at multiple network routers at extremely high rates, and queries are often posed on the union of data observed at all the routers. Since the data set is changing, the query results could also be changing continuously with time. This has motivated the *continuous, distributed, streaming model* [8]. In this model there are k physically distributed sites receiving high-volume local streams of data. These sites talk to a central coordinator, who has to continuously respond to queries over the union of all streams observed so far. The challenge is to minimize the communication between the different sites and the coordinator, while providing an accurate answer to queries at the coordinator at all times.

A fundamental problem in this setting is to obtain a random sample drawn from the union of all distributed streams. This generalizes the classic *reservoir sampling* problem (see, e.g., [15], where the algorithm is attributed to Waterman; see also [19]) to the setting of multiple distributed streams, and has applications to approximate query answering, selectivity estimation, and query planning. For example, in the case of network routers, maintaining a random sample from the union of the streams is valuable for network monitoring tasks involving the detection of global properties [13]. Other problems on distributed stream processing, including the estimation of the number of distinct elements [7,8] and heavy hitters [4,14,17,21], use random sampling as a primitive.

The study of sampling in distributed streams was initiated by Cormode *et al* [9]. Consider a set of k different streams observed by the k sites with the total number of current items in the union of all streams equal to n. The authors in [9] show how k sites can maintain a random sample of s items without replacement from the union of their streams using an expected $O((k+s)\log n)$ messages between the sites and the central coordinator. The memory requirement of the central coordinator is s machine words, and the time requirement is $O((k+s)\log n)$. The memory requirement of the remote sites is a single machine word with constant time per stream update. Cormode *et al.* also prove that the expected number of messages sent in any scheme is $\Omega(k + s\log(n/s))$. Each message is assumed to be a single machine word, which can hold an integer of magnitude $(kns)^{O(1)}$.

Notation. All logarithms are to the base 2 unless otherwise specified. Throughout the paper, when we use asymptotic notation, the variable that is going to infinity is n, and s and k are functions of n.

1.1 Our Results

Our main contribution is an algorithm for sampling without replacement from distributed streams, as well as a matching lower bound showing that the message complexity of our algorithm is optimal. A summary of our results and a comparison with earlier work is shown in Figure 1.

New Algorithm: We present an algorithm which uses an expected

$$O\left(\frac{k\log(n/s)}{\log(1+(k/s))}\right)$$

number of messages for continuously maintaining a random sample of size s from k distributed data streams of total size n. Notice that if $s < k/8$, this number is $O\left(\frac{k\log(n/s)}{\log(k/s)}\right)$, while if $s \geq k/8$, this number is $O(s\log(n/s))$.

The memory requirement in our protocol at the central coordinator is s machine words, and the time requirement is $O\left(\frac{k\log n/s}{\log(1+k/s)}\right)$. The former is the same as that in the protocol of [9], while the latter improves their $O((k+s)\log n)$ time requirement. The remote sites in our scheme store a single machine word and use constant time per stream update, which is clearly optimal.

Our result leads to a significant improvement in the message complexity in the case when k is large. For example, for the basic problem of maintaining a single random sample from the union of distributed streams ($s = 1$), our algorithm leads to a factor of $O(\log k)$ decrease in the number of messages sent in the system over the algorithm in [9].

Our algorithm is simple, and only requires the central coordinator to communicate with a site if the site initiates the communication. This is useful in a setting where a site may go offline, since it does not require the ability of a site to receive broadcast messages.

	Upper Bound		Lower Bound	
	Our Result	Cormode *et al.*	Our Result	Cormode *et al.*
$s < \frac{k}{8}$	$O\left(\frac{k\log(n/s)}{\log(k/s)}\right)$	$O(k\log n)$	$\Omega\left(\frac{k\log(n/s)}{\log(k/s)}\right)$	$\Omega(k + s\log n)$
$s \geq \frac{k}{8}$	$O(s\log(n/s))$	$O(s\log n)$	$\Omega(s\log(n/s))$	$\Omega(s\log(n/s))$

Fig. 1. Summary of Our Results for Message Complexity of Sampling Without Replacement

Lower Bound: We also show that for any constant $q > 0$, any correct protocol must send $\Omega\left(\frac{k\log(n/s)}{\log(1+(k/s))}\right)$ messages with probability at least $1 - q$. This also yields a bound of $\Omega\left(\frac{k\log(n/s)}{\log(1+(k/s))}\right)$ on the expected message complexity of any correct protocol, showing the expected number of messages sent by our algorithm is optimal, upto constant factors.

In addition to being quantitatively stronger than the lower bound of [9], our lower bound is also qualitatively stronger, because the lower bound in [9] is on the expected number of messages transmitted in a correct protocol. However, this does not rule out the possibility that with large probability, much fewer messages are sent in the optimal protocol. In contrast, we lower bound the number of messages that must be transmitted in any protocol 99% of the time. Since the time complexity of the central coordinator is at least the number of messages received, the time complexity of our protocol is also optimal.

Sampling with Replacement. We also show how to modify our protocol to obtain a random sample of s items from k distributed streams with replacement. Here we achieve a protocol with $O\left(\left(\frac{k}{\log(2+(k/(s\log s)))} + s\log s\right)\log n\right)$ messages, improving the $O((k + s\log s)\log n)$-message protocol of [9]. We obtain the same improvement in the time complexity of the central coordinator.

Heavy-Hitters. As a corollary, we obtain a protocol for estimating the heavy hitters in distributed streams with the best known message complexity. In this problem we would like to find a set H of items so that if an element e occurs at least an ε fraction of times in the union of the streams, then $e \in H$, and if e occurs less than an $\varepsilon/2$ fraction of times in union of the streams, then $e \notin H$. It is known that $O(\varepsilon^{-2}\log n)$ random samples suffice to estimate the set of heavy hitters with high probability, and the previous best algorithm [9] was obtained by plugging $s = O(\varepsilon^{-2}\log n)$ into a protocol for distributed sampling. We thus improve the message complexity from $O((k + \varepsilon^{-2}\log n)\log n)$ to

$O\left(\frac{k\log(\varepsilon n)}{\log(\varepsilon k)} + \varepsilon^{-2}\log(\varepsilon n)\log n\right)$. This can be significant when k is large compared to $1/\varepsilon$.

1.2 Related Work

In addition to work discussed above, other research in the continuous distributed streaming model includes estimating frequency moments and counting the number of distinct elements [7,8], and estimating the entropy [2]. The reservoir sampling technique has been used extensively in large scale data mining applications, see for example [10,16,1]. Stream sampling under sliding windows has been considered in [6,3]. Deterministic algorithms for heavy-hitters over distributed streams, and corresponding lower bounds were considered in [21].

Stream sampling under sliding windows over distributed streams has been considered in [9]. Their algorithm for sliding windows is already optimal upto lower-order additive terms (see Theorems 4.1 and 4.2 in [9]). Hence our improved results for the non-sliding window case do not translate into an improvement for the case of sliding windows.

A related model of distributed streams was considered in [11,12]. In this model, the coordinator was not required to continuously maintain an estimate of the required aggregate, but when the query was posed to the coordinator, the sites would be contacted and the query result would be constructed. In their model, the coordinator could be said to be "reactive", whereas in the model considered in this paper, the coordinator is "pro-active".

Roadmap: We first present the model and problem definition in Section 2, and then the algorithm followed by a proof of correctness in Section 3. The analysis of message complexity and the lower bound are presented in Sections 4 and 5 respectively, followed by an algorithm for sampling with replacement in Section 6.

2 Model

Consider a system with k different sites, numbered from 1 till k, each receiving a local stream of elements. Let $\mathcal{S}_i$ denote the stream observed at site i. There is one "coordinator" node, which is different from any of the sites. The coordinator does not observe a local stream, but all queries for a random sample arrive at the coordinator. Let $\mathcal{S} = \cup_{i=1}^{n}\mathcal{S}_i$ be the entire stream observed by the system, and let $n = |\mathcal{S}|$. The sample size s is a parameter supplied to the coordinator and to the sites during initialization.

The task of the coordinator is to continuously maintain a random sample $\mathcal{P}$ of size $\min\{n, s\}$ consisting of elements chosen uniformly at random without replacement from $\mathcal{S}$. The cost of the protocol is the number of messages transmitted.

We assume a synchronous communication model, where the system progresses in "rounds". In each round, each site can observe one element (or none), and send a message to the coordinator, and receive a response from the coordinator.

The coordinator may receive up to k messages in a round, and respond to each of them in the same round. This model is essentially identical to the model assumed in previous work [9]. Later we discuss how to handle the case of a site observing multiple elements per round.

The sizes of the different local streams at the sites, their order of arrival, and the interleaving of the streams at different sites, can all be arbitrary. The algorithm cannot make any assumption about these.

3 Algorithm

The idea in the algorithm is as follows. Each site associates a random "weight" with each element that it receives. The coordinator then maintains the set $\mathcal{P}$ of s elements with the minimum weights in the union of the streams at all times, and this is a random sample of $\mathcal{S}$. This idea is similar to the spirit in all centralized reservoir sampling algorithms. In a distributed setting, the interesting aspect is at what times do the sites communicate with the coordinator, and vice versa.

In our algorithm, the coordinator maintains u, which is the s-th smallest weight so far in the system, as well as the sample $\mathcal{P}$, consisting of all the elements that have weight no more than u. Each site need only maintain a single value u_i, which is the site's view of the s-th smallest weight in the system so far. Note that it is too expensive to keep the view of each site synchronized with the coordinator's view at all times – to see this, note that the value of the s-th smallest weight changes $O(s\log(n/s))$ times, and updating every site each time the s-th minimum changes takes a total of $O(sk\log(n/s))$ messages.

In our algorithm, when site i sees an element with a weight smaller than u_i, it sends it to the central coordinator. The coordinator updates u and $\mathcal{P}$, if needed, and then replies back to i with the current value of u, which is the true minimum weight in the union of all streams. Thus each time a site communicates with the coordinator, it either makes a change to the random sample, or, at least, gets to refresh its view of u.

The algorithm at each site is described in Algorithms 1 and 2. The algorithm at the coordinator is described in Algorithm 3.

Algorithm 1. Initialization at Site i

```
/* $u_i$ is site $i$'s view of the $s$-th smallest weight in the
   union of all streams so far. Note this may ``lag'' the
   value stored at the coordinator.                          */
$u_i \leftarrow 1$;
```

Algorithm 2. When Site i receives element e

```
Let $w(e)$ be a randomly chosen weight between 0 and 1;
if $w(e) < u_i$ then
    Send $(e, w(e))$ to the Coordinator and receive $u'$ from Coordinator;
    Set $u_i \leftarrow u'$;
```

Algorithm 3. Algorithm at Coordinator

```
/* The random sample $\mathcal{P}$ consists of tuples $(e, w)$ where $e$ is an
   element, and $w$ the weight, such that the weights are the $s$
   smallest among all the weights so far in the stream      */
$\mathcal{P} \leftarrow \phi$;
/* $u$ is the value of the $s$-th smallest weight in the stream
   observed so far. If there are less than $s$ elements so far,
   then $u$ is 1.                                           */
$u \leftarrow 1$;
while true do
    if a message $(e_i, u_i)$ arrives from site $i$ then
        if $u_i < u$ then
            Insert $(e_i, u_i)$ into $\mathcal{P}$;
            if $|\mathcal{P}| > s$ then
                Discard the element $(e, w)$ from $\mathcal{P}$ with the largest weight;
                Update $u$ to the current largest weight in $\mathcal{P}$ (which is also
                the $s$-th smallest weight in the entire stream);
        Send $u$ to site $i$;
    if a query for a random sample arrives then
        return $\mathcal{P}$
```

3.1 Correctness

The following two lemmas establish the correctness of the algorithm.

Lemma 1. *Let n be the number of elements in $\mathcal{S}$ so far. (1) If $n \leq s$, then the set $\mathcal{P}$ at the coordinator contains all the (e, w) pairs seen at all the sites so far. (2) If $n > s$, then $\mathcal{P}$ at the coordinator consists of the s (e, w) pairs such that the weights of the pairs in $\mathcal{P}$ are the smallest weights in the stream so far.*

The proof of this lemma has been omitted due to space constraints.

Lemma 2. *At the end of each round, sample $\mathcal{P}$ at the coordinator consists of a uniform random sample of size $\min\{n, s\}$ chosen without replacement from $\mathcal{S}$.*

Proof. In case $n \leq s$, then from Lemma 1, we know that $\mathcal{P}$ contains every element of $\mathcal{S}$. In case $n > s$, from Lemma 1, it follows that $\mathcal{P}$ consists of s elements with the smallest weights from $\mathcal{S}$. Since the weights are assigned randomly, each element in $\mathcal{S}$ has a probability of $\frac{s}{n}$ of belonging in $\mathcal{P}$, showing that this is an uniform random sample. Since an element can appear no more than once in the sample, this is a sample chosen without replacement. □

4 Analysis of the Algorithm (Upper Bound)

We now analyze the message complexity of the maintenance of a random sample.

For the sake of analysis, we divide the execution of the algorithm into "epochs", where each epoch consists of a sequence of rounds. The epochs are defined inductively. Let $r > 1$ be a parameter, which will be fixed later. Recall that u is the s-th smallest weight so far in the system (if there are fewer than s elements so far, $u = 1$). Epoch 0 is the set of all rounds from the beginning of execution until (and including) the earliest round where u is $\frac{1}{r}$ or smaller. Let m_i denote the value of u at the end of epoch $i-1$. Then epoch i consists of all rounds subsequent to epoch $i-1$ until (and including) the earliest round when u is $\frac{m_i}{r}$ or smaller. Note that the algorithm does not need to be aware of the epochs, and this is only used for the analysis.

Suppose we call the original distributed algorithm described in Algorithms 3 and 2 as Algorithm A. For the analysis, we consider a slightly different distributed algorithm, Algorithm B, described below. *Algorithm B is identical to Algorithm A except for the fact that at the beginning of each epoch, the value u is broadcast by the coordinator to all sites.*

While Algorithm A is natural, Algorithm B is easier to analyze. We first note that on the same inputs, the value of u (and $\mathcal{P}$) at the coordinator at any round in Algorithm B is identical to the value of u (and $\mathcal{P}$) at the coordinator in Algorithm A at the same round. Hence, the partitioning of rounds into epochs is the same for both algorithms, for a given input. The correctness of Algorithm B follows from the correctness of Algorithm A. The only difference between them is in the total number of messages sent. In B we have the property that for all i from 1 to k, $u_i = u$ at the beginning of each epoch (though this is not necessarily true throughout the epoch), and for this, B has to pay a cost of at least k messages in each epoch.

Lemma 3. *The number of messages sent by Algorithm A for a set of input streams $\mathcal{S}_j, j = 1 \ldots k$ is never more than twice the number of messages sent by Algorithm B for the same input.*

Proof. Consider site v in a particular epoch i. In Algorithm B, v receives m_i at the beginning of the epoch through a message from the coordinator. In Algorithm A, v may not know m_i at the beginning of epoch i. We consider two cases.

Case I: v sends a message to the coordinator in epoch i in Algorithm A. In this case, the first time v sends a message to the coordinator in this epoch, v will receive the current value of u, which is smaller than or equal to m_i. This communication costs two messages, one in each direction. Henceforth, in this epoch, the number of messages sent in Algorithm A is no more than those sent in B. In this epoch, the number of messages transmitted to/from v in A is at most twice the number of messages as in B, which has at least one transmission from the coordinator to site v.

Case II: v did not send a message to the coordinator in this epoch, in Algorithm A. In this case, the number of messages sent in this epoch to/from site v in Algorithm A is smaller than in Algorithm B. □

Let ξ denote the total number of epochs.

Lemma 4. *If* $r \geq 2$*,*

$$E[\xi] \leq \left(\frac{\log(n/s)}{\log r}\right) + 2$$

Proof. Let $z = \left(\frac{\log(n/s)}{\log r}\right)$. First, we note that in each epoch, u decreases by a factor of at least r. Thus after $(z + \ell)$ epochs, u is no more than $\frac{1}{r^{z+\ell}} = (\frac{s}{n})\frac{1}{r^\ell}$. Thus, we have

$$\Pr[\xi \geq z + \ell] \leq \Pr\left[u \leq \left(\frac{s}{n}\right)\frac{1}{r^\ell}\right]$$

Let Y denote the number of elements (out of n) that have been assigned a weight of $\frac{s}{nr^\ell}$ or lesser. Y is a binomial random variable with expectation $\frac{s}{r^\ell}$. Note that if $u \leq \frac{s}{nr^\ell}$, it must be true that $Y \geq s$.

$$\Pr[\xi \geq z + \ell] \leq \Pr[Y \geq s] \leq \Pr[Y \geq r^\ell E[Y]] \leq \frac{1}{r^\ell}$$

where we have used Markov's inequality.

Since ξ takes only positive integral values,

$$\begin{aligned} E[\xi] &= \sum_{i>0} \Pr[\xi \geq i] = \sum_{i=1}^{z} \Pr[\xi \geq i] + \sum_{\ell \geq 1} \Pr[\xi \geq z + \ell] \\ &\leq z + \sum_{\ell \geq 1} \frac{1}{r^\ell} \leq z + \frac{1}{1 - 1/r} \leq z + 2 \end{aligned}$$

where we have assumed $r \geq 2$. □

Let n_j denote the total number of elements that arrived in epoch j. We have $n = \sum_{j=0}^{\xi-1} n_j$. Let μ denote the total number of messages sent during the entire execution. Let μ_i denote the total number of messages sent in epoch i. Let X_i denote the number of messages sent from the sites to the coordinator in epoch i. μ_i is the sum of two parts, (1) k messages sent by the coordinator at the start of the epoch, and (2) two times the number of messages sent from the sites to the coordinator.

$$\mu_i = k + 2X_i \tag{1}$$

$$\mu = \sum_{j=0}^{\xi-1} \mu_i = \xi k + 2\sum_{j=0}^{\xi-1} X_j \tag{2}$$

Consider epoch i. For each each element $j = 1 \ldots n_i$ in epoch i, we define a 0-1 random variable Y_j as follows. $Y_j = 1$ if observing the j-th element in the epoch resulted in a message being sent to the coordinator, and $Y_j = 0$ otherwise.

$$X_i = \sum_{j=1}^{n_i} Y_j \tag{3}$$

Let $F(\eta, \alpha)$ denote the event $n_i = \eta$ and $m_i = \alpha$. The following Lemma gives a bound on a conditional probability that is used later.

Lemma 5. *For each $j = 1 \dots n_i - 1$*

$$\Pr[Y_j = 1 | F(\eta, \alpha)] \le \frac{\alpha - \alpha/r}{1 - \alpha/r}$$

Proof. Suppose that the j-th element in the epoch was observed by site v. For this element to cause a message to be sent to the coordinator, the random weight assigned to it must be less than u_v at that instant. Conditioned on $m_i = \alpha$, u_v is no more than α.

Note that in this lemma we exclude the last element that arrived in epoch i, thus the weight assigned to element j must be greater than α/r. Thus, the weight assigned to j must be a uniform random number in the range $(\alpha/r, 1)$. The probability this weight is less than the current value of u_v is no more than $\frac{\alpha-\alpha/r}{1-\alpha/r}$, since $u_v \le \alpha$. □

Lemma 6. *For each epoch i*

$$E[X_i] \le 1 + 2rs$$

Proof. We first obtain the expectation conditioned on $F(\eta, \alpha)$, and then remove the conditioning. From Lemma 5 and Equation 3 we get:

$$E[X_i|F(\eta,\alpha)] \le 1 + E\left[(\sum_{j=1}^{\eta-1} Y_j)|F(\eta,\alpha)\right] \le 1 + \sum_{j=1}^{\eta-1} E[Y_j|F(\eta,\alpha)] \le 1 + (\eta - 1)\frac{\alpha - \alpha/r}{1 - \alpha/r}.$$

Using $r \ge 2$ and $\alpha \le 1$, we get: $E[X_i|F(\eta, \alpha)] \le 1 + 2(\eta - 1)\alpha$.

We next consider the conditional expectation $E[X_i|m_i = \alpha]$.

$$\begin{aligned}
E[X_i|m_i = \alpha] &= \sum_{\eta} \Pr[n_i = \eta|m_i = \alpha] E[X_i|n_i = \eta, m_i = \alpha] \\
&\le \sum_{\eta} \Pr[n_i = \eta|m_i = \alpha](1 + 2(\eta - 1)\alpha) \\
&\le E[1 + 2(n_i - 1)\alpha|m_i = \alpha] \\
&\le 1 + 2\alpha(E[n_i|m_i = \alpha] - 1)
\end{aligned}$$

Using Lemma 7, we get

$$E[X_i|m_i = \alpha] \le 1 + 2\alpha\left(\frac{rs}{\alpha} - 1\right) \le 1 + 2rs$$

Since $E[X_i] = E[E[X_i|m_i = \alpha]]$, we have $E[X_i] \le E[1 + 2rs] = 1 + 2rs$. □

Lemma 7

$$E[n_i|m_i = \alpha] = \frac{rs}{\alpha}$$

Proof. Recall that n_i, the total number of elements in epoch i, is the number of elements observed till the s-th minimum in the stream decreases to a value that is less than or equal to α/r.

Let Z denote a random variable that equals the number of elements to be observed from the start of epoch i till s new elements are seen, each of whose weight is less than or equal to α/r. Clearly, conditioned on $m_i = \alpha$, it must be true that $n_i \le Z$. For $j = 1$ to s, let Z_j denote the number of elements observed from the state when $(j-1)$ elements have been observed with weights that are less than α/r till the state when j elements have been observed with weights less than α/r. Z_j is a geometric random variable with parameter α/r.

We have $Z = \sum_{j=1}^{s} Z_j$ and $E[Z] = \sum_{j=1}^{s} E[Z_j] = \frac{sr}{\alpha}$. Since $E[n_i|m_i = \alpha] \le E[Z]$, the lemma follows. □

Lemma 8

$$E[\mu] \le (k + 4rs + 2)\left(\frac{\log(n/s)}{\log r} + 2\right)$$

Proof. Using Lemma 6 and Equation 1, we get the expected number of messages in epoch i:

$$E[\mu_i] \le k + 2(2rs + 1) = k + 2 + 4rs$$

Note that the above is independent of i. The proof follows from Lemma 4, which gives an upper bound on the expected number of epochs. □

Theorem 1. *The expected message complexity $E[\mu]$ of our algorithm is as follows.*

I: If $s \ge \frac{k}{8}$, then $E[\mu] = O\left(s \log\left(\frac{n}{s}\right)\right)$
II: If $s < \frac{k}{8}$, then $E[\mu] = O\left(\frac{k \log\left(\frac{n}{s}\right)}{\log\left(\frac{k}{s}\right)}\right)$

Proof. We note that the upper bounds on $E[\mu]$ in Lemma 8 hold for any value of $r \ge 2$.
Case I: $s \ge \frac{k}{8}$. In this case, we set $r = 2$. From Lemma 8,

$$E[\mu] \le (8s + 8s + 2)\left(\frac{\log(n/s)}{\log 2}\right) = (16s + 2)\log\left(\frac{n}{s}\right) = O\left(s \log\left(\frac{n}{s}\right)\right)$$

Case II: $s < \frac{k}{8}$. We minimize the expression setting $r = \frac{k}{4s}$, and get: $E[\mu] = O\left(\frac{k \log\left(\frac{n}{s}\right)}{\log\left(\frac{k}{s}\right)}\right)$. □

5 Lower Bound

Theorem 2. *For any constant $q, 0 < q < 1$, any correct protocol must send $\Omega\left(\frac{k \log(n/s)}{\log(1+(k/s))}\right)$ messages with probability at least $1-q$, where the probability is taken over the protocol's internal randomness.*

Proof. Let $\beta = (1 + (k/s))$. Define $e = \Theta\left(\frac{\log(n/s)}{\log(1+(k/s))}\right)$ epochs as follows: in the i-th epoch, $i \in \{0, 1, 2, \ldots, e-1\}$, there are $\beta^{i-1}k$ global stream updates, which can be distributed among the k servers in an arbitrary way.

We consider a distribution on orderings of the stream updates. Namely, we think of a totally-ordered stream $1, 2, 3, \ldots, n$ of n updates, and in the i-th epoch, we randomly assign the $\beta^{i-1}k$ updates among the k servers, independently for each epoch. Let the randomness used for the assignment in the i-th epoch be denoted σ_i.

Consider the global stream of updates $1, 2, 3, \ldots, n$. Suppose we maintain a sample set $\mathcal{P}$ of s items without replacement. We let $\mathcal{P}_i$ denote a random variable indicating the value of $\mathcal{P}$ after seeing i updates in the stream. We will use the following lemma about reservoir sampling.

Lemma 9. *For any constant $q > 0$, there is a constant $C' = C'(q) > 0$ for which*

- *$\mathcal{P}$ changes at least $C's\log(n/s)$ times with probability at least $1-q$, and*
- *If $s < k/8$ and $k = \omega(1)$ and $e = \omega(1)$, then with probability at least $1 - q/2$, over the choice of $\{\mathcal{P}_i\}$, there are at least $(1 - (q/8))e$ epochs for which the number of times $\mathcal{P}$ changes in the epoch is at least $C's\log(1 + (k/s))$.*

Proof. Consider the stream $1, 2, 3, \ldots, n$ of updates. In the classical reservoir sampling algorithm [15], $\mathcal{P}$ is initialized to $\{1, 2, 3, \ldots, s\}$. Then, for each $i > s$, the i-th element is included in the current sample set $\mathcal{P}_i$ with probability s/i, in which case a random item in $\mathcal{P}_{i-1}$ is replaced with i.

For the first part of Lemma 9, let X_i be an indicator random variable if i causes $\mathcal{P}$ to change. Let $X = \sum_{i=1}^{n} X_i$. Hence, $\mathbf{E}[X_i] = 1$ for $1 \le i \le s$, and $\mathbf{E}[X_i] = s/i$ for all $i > s$. Then $\mathbf{E}[X] = s + \sum_{i=s+1}^{n} s/i = s + s(H_n - H_s)$, where $H_i = \ln i + O(1)$ is the i-th Harmonic number. Then all of the X_i are independent indicator random variables. It follows by a Chernoff bound that

$$\begin{aligned} \Pr[X < \mathbf{E}[X]/2] &\le \exp(-\mathbf{E}[X]/8) \\ &\le \exp(-(s + s\ln(n/s) - O(1))/8) \\ &\le \exp(-\Theta(s))\left(\frac{s}{n}\right)^{s/8}. \end{aligned}$$

There is an absolute constant n_0 so that for any $n \ge n_0$, this probability is less than any constant q, and so the first part of Lemma 9 follows.

For the second part of Lemma 9, consider the i-th epoch, $i > 0$, which contains $\beta^{i-1}k$ consecutive updates. Let Y_i be the number of changes in this epoch. Then

$\mathbf{E}[Y_i] \geq s(H_{\beta^{i-1}k} - H_{\beta^{i-2}k}) = \Omega(s \log \beta)$. Note that $\beta \geq 9$ since $s < k/8$ by assumption, and $\beta = 1 + k/s$. Since Y_i can be written as a sum of independent indicator random variables, by a Chernoff bound,

$$\Pr[Y_i < \mathbf{E}[Y_i]/2] \leq \exp(-\mathbf{E}[Y_i]/8) \leq \exp(-\Omega(s \log \beta)) \leq \frac{1}{\beta^{\Omega(s)}}.$$

Hence, the expected number of epochs i for which $Y_i < \mathbf{E}[Y_i]/2$ is at most $\sum_{i=1}^{e-1} \frac{1}{\beta^{\Omega(s)}}$, which is $o(e)$ since $\beta \geq 9$ and $e = \omega(1)$. By a Markov bound, with probability at least $1 - q/2$, at most $o(e/q) = o(e)$ epochs i satisfy $Y_i < \mathbf{E}[Y_i]/2$. It follows that with probability at least $1 - q/2$, there are at least $(1 - q/8)e$ epochs i for which the number Y_i of changes in the epoch i is at least $\mathbf{E}[Y_i]/2 \geq C' s \log \beta = C' s \log(1 + (k/s))$ for a constant $C' > 0$, as desired.

□

Corner Cases: When $s \geq k/8$, the statement of Theorem 2 gives a lower bound of $\Omega(s \log(n/s))$. In this case Theorem 2 follows immediately from the first part of Lemma 9 since these changes in $\mathcal{P}$ must be communicated to the central coordinator. Hence, in what follows we can assume $s < k/8$. Notice also that if $k = O(1)$, then $\frac{k \log(n/s)}{\log(1+(k/s))} = O(s \log(n/s))$, and so the theorem is independent of k, and follows simply by the first part of Lemma 9. Notice also that if $e = O(1)$, then the statement of Theorem 2 amounts to proving an $\Omega(k)$ lower bound, which follows trivially since every site must send at least one message.

Thus, in what follows, we may apply the second part of Lemma 9.

Main Case: Let $C > 0$ be a sufficiently small constant, depending on q, to be determined below. Let Π be a possibly randomized protocol, which with probability at least q, sends at most Cke messages. We show that Π cannot be a correct protocol.

Let τ denote the random coin tosses of Π, i.e., the concatenation of random strings of all k sites together with that of the central coordinator.

Let $\mathcal{E}$ be the event that Π sends less than Cke messages. By assumption, $\Pr_\tau[\mathcal{E}] \geq q$. Hence, it is also the case that

$$\Pr_{\tau, \{\mathcal{P}_i\}, \{\sigma_i\}}[\mathcal{E}] \geq q.$$

For a sufficiently small constant $C' > 0$ that may depend on q, let $\mathcal{F}$ be the event that there are at least $(1 - (q/8))e$ epochs for which the number of times $\mathcal{P}$ changes in the epoch is at least $C' s \log(1 + (k/s))$. By the second part of Lemma 9,

$$\Pr_{\tau, \{\mathcal{P}_i\}, \{\sigma_i\}}[\mathcal{F}] \geq 1 - q/2.$$

It follows that there is a fixing of $\tau = \tau'$ as well as a fixing of $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_e$ to $P'_0, P'_1, \ldots, P'_e$ for which $\mathcal{F}$ occurs and

$$\Pr_{\{\sigma_i\}}[\mathcal{E} \mid \tau = \tau', \ (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_e) = (P'_0, P'_1, \ldots, P'_e)] \geq q - q/2 = q/2.$$

Notice that the three (sets of) random variables $\tau, \{P_i\}$, and $\{\sigma_i\}$ are independent, and so in particular, $\{\sigma_i\}$ is still uniformly random given this conditioning.

By a Markov argument, if event $\mathcal{E}$ occurs, then there are at least $(1-(q/8))e$ epochs for which at most $(8/q)\cdot C\cdot k$ messages are sent. If events $\mathcal{E}$ and $\mathcal{F}$ both occur, then by a union bound, there are at least $(1-(q/4))e$ epochs for which at most $(8/q)\cdot C\cdot k$ messages are sent and $\mathcal{P}$ changes in the epoch at least $C's\log(1+(k/s))$ times. Call such an epoch *balanced.*

Let i^* be the epoch which is most likely to be balanced, over the random choices of $\{\sigma_i\}$, conditioned on $\tau=\tau'$ and $(\mathcal{P}_0,\mathcal{P}_1,\ldots,\mathcal{P}_e)=(P'_0,P'_1,\ldots,P'_e)$. Since at least $(1-(q/4))e$ epochs are balanced if $\mathcal{E}$ and $\mathcal{F}$ occur, and conditioned on $(\mathcal{P}_0,\mathcal{P}_1,\ldots,\mathcal{P}_e)=(P'_0,P'_1,\ldots,P'_e)$ event $\mathcal{F}$ does occur, and $\mathcal{E}$ occurs with probability at least $q/2$ given this conditioning, it follows that

$$\Pr_{\{\sigma_i\}}[i^* \text{ is balanced } \mid \tau=\tau',\ (\mathcal{P}_0,\mathcal{P}_1,\ldots,\mathcal{P}_e)=(P'_0,P'_1,\ldots,P'_e)] \geq q/2-q/4=q/4.$$

The property of i^* being balanced is independent of σ_j for $j\neq i^*$, so we also have

$$\Pr_{\sigma_{i^*}}[i^* \text{ is balanced } \mid \tau=\tau',\ (\mathcal{P}_0,\mathcal{P}_1,\ldots,\mathcal{P}_e)=(P'_0,P'_1,\ldots,P'_e)] \geq q/4.$$

Since $C's\log(1+(k/s))>0$ and $\mathcal{P}$ changes at least $C's\log(1+(k/s))$ times in epoch i^*, we have that $\mathcal{P}$ changes at least once in epoch i^*. Suppose the first update in the global stream at which $\mathcal{P}$ changes is the j^*-th update. In order for i^* to be balanced for at least a $q/4$ fraction of the σ_{i^*}, there must be at least $qk/4$ different servers which receive j^*, for which Π sends a message. In particular, since Π is deterministic conditioned on τ, at least $qk/4$ messages must be sent in the i^*-th epoch. But i^* was chosen so that at most $(8/q)\cdot C\cdot k$ messages are sent, which is a contradiction for $C<q^2/32$.

It follows that we have reached a contradiction, and so it follows that Cke messages must be sent with probability at least $1-q$. Since $Cke=\Omega\left(\frac{k\log(n/s)}{\log(1+(k/s))}\right)$, this completes the proof. □

6 Sampling with Replacement

We now present an algorithm to maintain a random sample of size s with replacement from $\mathcal{S}$. The basic idea is to run in parallel s copies of the single item sampling algorithm from Section 3. Done naively, this will lead to a message complexity of $O(sk\frac{\log n}{\log k})$. We obtain an improved algorithm based on the following ideas.

We view the distributed streams as s logical streams, $\mathcal{S}^i, i=1\ldots s$. Each $\mathcal{S}^i$ is identical to $\mathcal{S}$, but the algorithm assigns independent weights to the different copies of the same element in the different logical streams. Let $w^i(e)$ denote the weight assigned to element e in $\mathcal{S}^i$. $w^i(e)$ is a random number between 0 and 1. For each $i=1\ldots s$, the coordinator maintains the minimum weight, say w^i, among all elements in $\mathcal{S}^i$, and the corresponding element.

Let $\beta = \max_{i=1}^{s} w^i$; β is maintained by the coordinator. Each site j maintains β_j, a local view of β, which is always greater than or equal to β. Whenever a logical stream element at site j has weight less than β_j, the site sends it to the coordinator, receives in response the current value of β, and updates β_j. When a random sample is requested at the coordinator, it returns the set of all minimum weight elements in all s logical streams. It can be easily seen that this algorithm is correct, and at all times, returns a random sample of size s selected with replacement. The main optimization relative to the naive approach described above is that when a site sends a message to the coordinator, it receives β, which provides partial information about all w^is. This provides a substantial improvement in the message complexity and leads to the following bounds.

Theorem 3. *The above algorithm continuously maintains a sample of size s with replacement from $\mathcal{S}$, and its expected message complexity is $O(s \log s \log n)$ in case $k \leq 2s \log s$, and $O\left(k \frac{\log n}{\log(\frac{k}{s \log s})}\right)$ in case $k > 2s \log s$.*

Proof. We provide a sketch of the proof here. The analysis of the message complexity is similar to the case of sampling without replacement. We sketch the analysis here, and omit the details. The execution is divided into epochs, where in epoch i, the value of β at the coordinator decreases by at least a factor of r (a parameter to be determined later). Let ξ denote the number of epochs. It can be seen that $E[\xi] = O(\frac{\log n}{\log r})$. In epoch i, let X_i denote the number of messages sent from the sites to the coordinator in the epoch, m_i denote the value of β at the beginning of the epoch, and n_i denote the number of elements in $\mathcal{S}$ that arrived in the epoch.

The n_i elements in epoch i give rise to sn_i logical elements, and each logical element has a probability of no more than m_i of resulting in a message to the coordinator. Similar to the proof of Lemma 6, we can show using conditional expectations that $E[X_i] \leq rs \log s$ (the $\log s$ factor comes in due to the fact that $E[n_i | m_i = \alpha] \leq \frac{r \log s}{\alpha}$. Thus the expected total number of messages in epoch i is bounded by $(k + 2sr \log s)$, and in the entire execution is $O((k + 2sr \log s)\frac{\log n}{\log r})$. By choosing $r = 2$ for the case $k \leq (2s \log s)$, and $r = k/(s \log s)$ for the case $k > (2s \log s)$, we get the desired result. □

References

1. Aggarwal, C.C.: On biased reservoir sampling in the presence of stream evolution. In: VLDB, pp. 607–618 (2006)
2. Arackaparambil, C., Brody, J., Chakrabarti, A.: Functional monitoring without monotonicity. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part 1. LNCS, vol. 5555, pp. 95–106. Springer, Heidelberg (2009)
3. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In: SODA, pp. 633–634 (2002)
4. Babcock, B., Olston, C.: Distributed top-k monitoring. In: SIGMOD Conference, pp. 28–39 (2003)

5. Braverman, V., Ostrovsky, R.: Effective computations on sliding windows. SIAM Journal on Computing 39(6), 2113–2131 (2010)
6. Braverman, V., Ostrovsky, R., Zaniolo, C.: Optimal sampling from sliding windows. In: PODS, pp. 147–156 (2009)
7. Cormode, G., Garofalakis, M.N.: Sketching streams through the net: Distributed approximate query tracking. In: VLDB, pp. 13–24 (2005)
8. Cormode, G., Muthukrishnan, S., Yi, K.: Algorithms for distributed functional monitoring. In: SODA, pp. 1076–1085 (2008)
9. Cormode, G., Muthukrishnan, S., Yi, K., Zhang, Q.: Optimal sampling from distributed streams. In: PODS, pp. 77–86 (2010)
10. Dash, M., Ng, W.: Efficient reservoir sampling for transactional data streams. In: Sixth IEEE International Conference on Data Mining (Workshops), pp. 662–666 (2006)
11. Gibbons, P.B., Tirthapura, S.: Estimating simple functions on the union of data streams. In: SPAA, pp. 281–291 (2001)
12. Gibbons, P.B., Tirthapura, S.: Distributed streams algorithms for sliding windows. In: SPAA, pp. 63–72 (2002)
13. Huang, L., Nguyen, X., Garofalakis, M.N., Hellerstein, J.M., Jordan, M.I., Joseph, A.D., Taft, N.: Communication-efficient online detection of network-wide anomalies. In: INFOCOM, pp. 134–142 (2007)
14. Keralapura, R., Cormode, G., Ramamirtham, J.: Communication-efficient distributed monitoring of thresholded counts. In: SIGMOD Conference, pp. 289–300 (2006)
15. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Seminumerical Algorithms, vol. II. Addison-Wesley, Reading (1981)
16. Malbasa, V., Vucetic, S.: A reservoir sampling algorithm with adaptive estimation of conditional expectation. In: IJCNN 2007, International Joint Conference on Neural Networks, pp. 2200–2204 (2007)
17. Manjhi, A., Shkapenyuk, V., Dhamdhere, K., Olston, C.: Finding (recently) frequent items in distributed data streams. In: ICDE, pp. 767–778 (2005)
18. Muthukrishnan, S.: Data Streams: Algorithms and Applications. In: Foundations and Trends in Theoretical Computer Science, Now Publishers (August 2005)
19. Vitter, J.S.: Random sampling with a reservoir. ACM Transactions on Mathematical Software 11(1), 37–57 (1985)
20. Xu, B., Tirthapura, S., Busch, C.: Sketching asynchronous data streams over sliding windows. Distributed Computing 20(5), 359–374 (2008)
21. Yi, K., Zhang, Q.: Optimal tracking of distributed heavy hitters and quantiles. In: PODS, pp. 167–174 (2009)

Parsimonious Flooding in Geometric Random-Walks *
(Extended Abstract)

Andrea E.F. Clementi[1] and Riccardo Silvestri[2]

[1] Dipartimento di Matematica, Università di Roma "Tor Vergata"
clementi@mat.uniroma2.it

[2] Dipartimento di Informatica, Università di Roma "La Sapienza"
silvestri@di.uniroma1.it

Abstract. We study the information spreading yielded by the *(Parsimonious) k-Flooding Protocol* in geometric Mobile Ad-Hoc Networks. We consider n agents on a square of side length L performing independent random walks with move radius ρ. At any time step, every active agent v informs every non-informed agent which is within distance R from v ($R > 0$ is the transmission radius). An agent is only active for the next k time steps following the one in which has been informed and, after that, she is removed. At the initial time step, a source agent is informed and we look at the *completion time* of the protocol, i.e., the first time step (if any) in which all agents are informed.

The presence of removed agents makes this process much more complex than the (standard) flooding and no analytical results are available over any explicit mobility model.

We prove optimal bounds on the completion time depending on the parameters n, L, R, and ρ. The obtained bounds hold with high probability. Our method of analysis provides a clear picture of the dynamic shape of the information spreading (or infection wave) over the time.

1 Introduction

In the *Geometric Random-Walk Model* [7,11,12], n agents perform independent random walks over a square of side length L and we consider the following *infection* process. Every agent can be in three different states: non-informed (white) state, informed-active (red), informed-removed (black). During a time step, every agent performs one step of the random walk and every red agent informs (infects) all white agents lying within distance R. A white agent that has been informed (for the first time) at time step t, she becomes red at time step $t+1$. Finally, when an agent becomes red she stays red for k time step and, after that, she becomes black and stays that, forever.

At the initial time step, a source agent is in the red state. The completion time of the above infection process is the first time step in which every agent

* Partially supported by the Italian MIUR-COFIN *COGENT*.

gets into the black state. If this time step does not exist then we say the infection process does not complete.

This random process is inspired by two main scenarios: The *Susceptible-Infected-Removed (SIR)* process which is widely studied in Mathematical Epidemiology [1,2,4] and the *(Parsimonious) k-Flooding Protocol* [3] in *geometric Mobile Ad-Hoc Networks (MANET).*

While the standard flooding is extremely inefficient in terms of agent's energy consumption and message complexity, the k-flooding protocol (for small values of k) strongly reduces the agent's energy consumption and the overall message complexity. However, as discussed later, the infection process yielded by the k-flooding (for small k) over a dynamic network is much more complex than that yielded by the standard flooding.

The k-flooding has been studied in [3] on *Edge-Markovian Evolving Graphs* (*Edge-MEG*). An Edge-MEG [6,7] is a Markovian random process that generates an infinite sequence of graphs over the same set of n agents. If an edge exists at time t then, at time $t+1$, it dies with probability q. If instead the edge does not exist at time t, then it will come into existence at time $t+1$ with probability p. The stationary distribution of an Edge-MEG with parameter p and q is the famous Erdös-Rényi random graph $\mathcal{G}(n, \tilde{p})$ where $\tilde{p} = \frac{p}{p+q}$. The work [3] gives tight bounds on the k-flooding on stationary Edge-MEG for arbitrary values of p, q and k. In particular, it derives the *reachability threshold* for the k-flooding, i.e., the smallest $k = k(n,p,q)$ over which the protocol completes.

Edge-MEG is an analytical model of dynamic networks capturing time-dependencies, an important feature observed in real scenarios such as (faulty) wireless networks and P2P networks. However, it does not model important features of MANET. Indeed, in Edge-MEG, edges are independent Markov chains while, in MANET, there is a strong correlation among edges: agents use to act over a geometric space [7,11,12,15].

Our Contribution. We study the k-flooding protocol in the geometric random-walk model (the resulting MANET will be called *geometric-MANET*). The move radius ρ determines the maximal distance an agent can travel in one time step. Even though network modelling issues are out of the aims of this work, both the transmission radius and the move radius play a crucial role in our analysis and, thus, a small discussion about them is needed. Both parameters depend on several factors. In mathematical epidemiology, they depend on the agent's mobility, the kind of infection and on the agent's social behaviour that all together determine the average rate of "positive" contacts. In typical biological cases, the move radius can be significantly larger than the transmission radius. In MANET, the move radius depends on, besides the agent's mobility, the adopted protocol that tunes the transmission rate of the agents: the larger is the time between two consecutive transmissions the larger is ρ. A larger ρ could yield a better message complexity at the cost of a larger completion time (the correct trade-offs is derived in [8] for the standard flooding).

It turns out that the issue of setting the "most suitable" move radius concerns both the network modelling and the protocol design. For these reasons, we investigate the k-flooding for a wide range of values for parameters R and ρ.

We first show a negative result. If the transmission radius R is below the *connectivity threshold* (the connectivity threshold refers to the uniform distribution of n (static) agents over the square), then for any $\rho \geqslant 0$, *with high probability (w.h.p.)*[1], the k-flooding does not complete for any $k = O(1)$.

We thus study the k-flooding for R over the connectivity threshold. We emphasize that the "static" connectivity among generic agents says nothing about the behaviour of red agents over the time which is the crucial issue in the k-flooding process.

If $\rho \lesssim R$, we prove that, for any choice of $k \geqslant 1$, the information spreads at "optimal" speed $\Theta(R)$, i.e., the k-flooding protocol w.h.p. completes within $O(L/R)$ time steps. Observe that, since $\rho \lesssim R$, this bound is asymptotically optimal.

Then, we consider move radii that can be up to any polynomial of R, i.e. $\rho \leqslant \text{POLY}(R)$. We prove that the information spreads at "optimal" speed $\Theta(\rho)$. So, for any $k \geqslant 1$, the k-flooding w.h.p. completes in time $O(L/\rho)$ which is optimal for any $\rho \geqslant R$. Notice that this optimal information speed makes the 1-flooding time smaller than the static diameter $O(L/R)$ of the stationary graph: our bound is thus the first analytical evidence that agent's mobility actually speeds-up this infection process. Finally, we observe that, in both cases, the energy-efficient 1-flooding protocol is as fast as the standard flooding [7,8].

Adopted Techniques and Further Results. It is important to emphasize that the presence of black agents in the infection process makes the analysis techniques adopted for the flooding almost useless. In particular, percolation theory [12,11,17], meeting and cover time of random walks on graphs [18], and the expansion/bootstrap arguments [7,8] strongly rely on the fact that an informed agent will be active for all the flooding process. Furthermore, the analysis of k-flooding over the Edge-MEG model [3] strongly relies on the stochastic independence among the edges and the consequent high node-expansion property of $\mathcal{G}(n,p)$: properties that clearly do not hold in geometric-MANET.

Our method of analysis significantly departs from all those mentioned above. Besides the optimal bounds on the completion time, our analyses provide a clear characterization of the geometric evolution of the infection process. We make use of a grid partition of the square into cells of size $\Theta(R)$ and define a set of possible states a cell can assume over time depending on the number of red, white and black agents inside it[2]. We then derive the local state-evolution

[1] An event is said to hold *with high probability (w.h.p.)* if its probability is at least $1-1/n$. For the sake of simplicity, we here adopt a weaker definition: the event must hold with probability at least $1 - 1/n^a$ for some constant $a > 0$. It is easy to verify that we can let all our results to hold with the standard high probability by just changing the constant factors in the analysis.

[2] When $\rho >> R$, we also need a further grid partition into *supercells* of size $\Theta(\rho)$ and a more complex argument.

law of any cell. Thanks to the regularity of this law, we can characterize the evolution of the geometric wave formed by the *red cells* (i.e. cells containing some red agent). A crucial property we prove is that, at any time step, white cells (i.e. cells containing white agents only) will never be adjacent to black cells, so there is always a red wave working between the black region and the white one. Furthermore, we show that the red wave eventually spans the entire region before all agents become black.

The generality of our method of analysis has further consequences. Thanks to the regularity of the red-wave shape, we are able to bound the time by which a given subregion will be infected for the first time. This bound is a function of the distance between the subregion and the initial position of the source agent. Actually, it is a function of the distance from the *closest* red agent at the starting time. So, our technique also works in the presence of an arbitrary set of source agents that aim to spread the same infection (or message). Under the same assumptions made for the single-source case, we can prove the completion time is w.h.p. $\Theta(\textsc{ecc}(A)/R)$ (or $\Theta(\textsc{ecc}(A)/\rho)$) where A is the set of the positions of the source agents at starting time and $\textsc{ecc}(A)$ is the *geometric eccentricity* of A in the square.

Related Works. As mentioned above, there are no analytical studies of the parsimonious flooding process over any geometric mobility model. In what follows, we briefly discuss some analytical results concerning the flooding over some models of MANET. In [14], the flooding time is studied over a restricted geometric-MANET. Approximately, it corresponds to the case $\rho = \Theta(L)$. Notice that, under this restriction, the stochastic dependence between two consecutive agent positions is negligible. In [13], the speed of data communication between two agents is studied over a class of *Random-Direction* models yielding uniform stationary agent's distributions (including the geometric-MANET model). They provide an upper bound on this speed that can be interpreted as a *lower* bound on flooding time when the mobile network is very sparse and disconnected (i.e. $R, \rho = o(1)$). Their technique, based on Laplacian transform of independent journeys, cannot be extended to provide any upper bound on the time of any version of the flooding. We observe that, differently from our model, both [14] and [13] assume that when an agent gets informed then all agents of her *current* connected component (no matter how large it is) will get informed in one time step. The same unrealistic assumption is adopted in [17], where bounds on flooding time (and some other tasks) are obtained in the Poisson approximation of the geometric-MANET model. In [7,8], the first almost tight bounds for the flooding time over geometric-MANET have been given. As mentioned before, their proofs strongly rely on the fact that informed agents stay always active. Flooding and gossip time for random walks over the grid graph (so, the agent's space is a graph) have been studied in [18]. Here, an agent informs all agents lying in the same node. Besides the differences between k-flooding and flooding discussed before, it is not clear whether their results could be extended to the random walk model over geometric spaces. Especially, in their model, there is no way to consider arbitrary values of the move radius.

Roadmap of the Paper. The rest of the paper is organized as follows. After giving some preliminary definitions in Section 2, we analyze the case $\rho \lesssim R$ in Section 3. In Section 4, we present the results for the case $\rho \leqslant R^2/\sqrt{\log n}$. In Section 5, we introduce a slight different geometric random-walk model and a more powerful technique to extend the bound $O(L/\rho)$ to any $\rho \leqslant \textsc{poly}(R)$. In Section 6, we first briefly describe the extension to multi-source case and, then, derive the completion threshold of R for the k-flooding, for constant k. Finally, some open questions are discussed in Section 7. Due to the lack of space, most of the proofs are omitted (see the full version of the paper [10].

2 Preliminaries

We analyze geometric-MANET over a square $\mathcal{Q}$ of side length $L > 0$ by considering some suitable partition $\mathcal{C}(\mathcal{Q})$ of $\mathcal{Q}$ into a grid of square *cells* of side length ℓ.

We say that two cells are adjacent if they touch each other by side or by corner. The *cell-diameter* $D_\Box(\mathcal{Q})$ of $\mathcal{Q}$ is defined as follows. Given two cells $\textsc{c}, \textsc{c}' \in \mathcal{C}(\mathcal{Q})$, define their *cell-distance* $d_\Box(\textsc{c}, \textsc{c}')$ as the length of the shortest cell-path $p = \langle \textsc{c} = \textsc{c}_0, \textsc{c}_1, \ldots, \textsc{c}_s = \textsc{c}' \rangle$ such that, for every i, $\textsc{c}_i$ is adjacent to $\textsc{c}_{i+1}$. Then, we define

$$D_\Box(\mathcal{Q}) \; = \; \max\{ \; d_\Box(\textsc{c}, \textsc{c}') \mid \textsc{c}, \textsc{c}' \in \mathcal{C}(\mathcal{Q}) \; \}$$

Similarly, we can define the *cell-distance* between a cell and any cell subset $\mathcal{C}$, i.e.,

$$d_\Box(\textsc{c}, \mathcal{C}) \; = \; \min\{ d_\Box(\textsc{c}, \textsc{c}') \mid \textsc{c}' \in \mathcal{C}\}$$

Observe that the size of the square and the *cell-diameter* $D_\Box(\mathcal{Q})$ are tightly related: $D_\Box(\mathcal{Q}) = \Theta(L/\ell)$.

According to the geometric random-walk model, there are n agents that perform independent random walks over $\mathcal{Q}$. At any time step, an agent in position $x \in \mathcal{Q}$, can move uniformly at random to any position in $\mathcal{B}(x, \rho) \cap \mathcal{Q}$, where $\mathcal{B}(x, \rho)$ is the disk of center x and radius ρ. This is a special case of the *Random Trip Model* introduced in [15] where it is proved that it admits a unique stationary agents distribution and, in this case, it is almost uniform. In the sequel, we always assume that at time $t = 0$ agents' positions are random w.r.t. the stationary distribution.

Let us consider n mobile agents acting over the square $\mathcal{Q}$ according to the geometric random-walk model. We say that the resulting geometric-MANET satisfies the *density property* w.r.t. the cell partition of side length ℓ if, with probability at least $1 - (1/n)^4$, for every time step $t = 0, 1, \ldots, n$ and for every cell $\textsc{c} \in \mathcal{C}(\mathcal{Q})$, the number $\#_\textsc{c}$ of agents in $\textsc{c}$ at time step t satisfies the following inequalities

$$[\textit{Density Property}] \qquad \eta_1 \ell^2 \; \leqslant \; \#_\textsc{c} \; \leqslant \; \eta_2 \ell^2, \text{where } \eta_1, \eta_2 \text{ are positive constants.} \tag{1}$$

We observe that a sufficient condition for the density property in our model is $\ell \geqslant \beta(L/\sqrt{n})\sqrt{\log n}$, for a suitable constant $\beta > 0$.

In the k-flooding protocol, at any time step, every agent can be in three different states: white (non-informed) state, red (informed-active), and black (informed-inactive). Then, the configuration $\textsc{conf}\,(t)$ is defined as the set of the positions of the n agents together with their respective states at the end of time t.

The analysis of the information spreading will be performed in terms of the number of infected cells. Given a cell $\mathrm{c} \in \mathcal{C}(\mathcal{Q})$, the neighborhood $N(\mathrm{c})$ is the set of cells formed by c and all its adjacent cells. Since the stationary agent's distribution is almost uniform and the maximal speed of any message in the geometric-MANET is $R + \rho$. we easily get the following lower bound.

Fact 1 *For any k, the k-flooding time is w.h.p. $\Omega(L/(R+\rho))$.*

For the sake of simplicity, we will prove our results by assuming $L = \sqrt{n}$ (i.e., average agent density equal to 1). It is straightforward to scale all the definitions and results to an arbitrary average density of agents. Furthermore, all the proofs of the upper bounds in Sections 3-5 are given for the case $k = 1$ (1-flooding), however, all our arguments can be easily extended to the k-flooding protocol, for any $k \geqslant 1$. Indeed, all our arguments concern the properties of the geometric wave formed by those red cells containing agents that have been *just* informed. So, the behaviour of the other red agents is not relevant in our analysis.

3 High Transmission-Rate or Low-Mobility

We warm-up with the case where the move radius is smaller than the transmission radius. More precisely, we assume $\rho \leqslant R/(2\sqrt{2})$. We consider a grid partition of $\mathcal{Q}$ into cells of side length $\ell = R/(2\sqrt{2})$ and the move radius satisfies $\rho \leq R/(2\sqrt{2})$. So, an agent lying in a cell c cannot escape from $N(\mathrm{c})$ in one step. This makes the analysis simpler than the cases with larger ρ. However, some of the crucial ideas and arguments adopted here will be exploited for the harder cases too.

Theorem 2. *Let $R \geqslant c_0 L\sqrt{\log n/n}$ for a sufficiently large constant c_0 and $\rho \leq R/(2\sqrt{2})$. Then, for any $k \geqslant 1$, the k-flooding time is w.h.p. $\Theta(L/R)$.*

3.1 Proof of Theorem 2

The lower bound is an immediate consequence of Fact 1. Let us now consider the upper bound. For the sake of simplicity, we assume that the time step is divided into 2 consecutive phases: the *transmission phase* where every red agent transmits the information and the *move phase* where every agent performs one step of the random walk.

We need to introduce the feasible states of a cell during the infection process.

Definition 1. *At (the end of) any time step a cell* $\textsc{c}$ *can be in 4 different states. It is white if it contains only white agents. It is red if it contains at least one red agent. It is black if it contains black agents only. It is grey if it is not in any of the previous 3 cases.*

We will show that the infection process, at any time step, has (w.h.p.) a well defined shape in which no white cell is adjacent to a black one and there is no grey cell. This shape is formalized in the following definitions. A subset of white cells is a *white component* if it is a connected component w.r.t. the subset of all the white cells.

Definition 2. *A configuration* $\textsc{conf}\,(t)$ *is* regular *if the following properties hold*

a) No cell is grey.
b) Every white component is adjacent to a red cell.
c) No white cell is adjacent to a black cell.

Observe that the starting configuration $\textsc{conf}\,(0)$ is regular.

Definition 3. *A white cell is red-close if it is adjacent to a red cell.*

The next lemma determines the local state-evolution of any cell in a regular configuration.

Lemma 1. *Consider any cell* $\textsc{c}$ *at time step* t *and assume* $\textsc{conf}\,(t)$ *be regular. Then the following properties hold.*
1) If $\textsc{c}$ *is red-close then it becomes red in* $\textsc{conf}\,(t+1)$ *w.h.p.*
2) If $\textsc{c}$ *is (no red-close) white then it becomes white or red in* $\textsc{conf}\,(t+1)$*.*
3) If $\textsc{c}$ *is red then it becomes red or black in* $\textsc{conf}\,(t+1)$*.*
4) If $\textsc{c}$ *is black then it becomes red or black in* $\textsc{conf}\,(t+1)$*.*

As an easy consequence of the above lemma, we get the following

Lemma 2. *For any* $t \leqslant n$*, if* $\textsc{conf}\,(t)$ *is regular, then w.h.p.* $\textsc{conf}\,(t+1)$ *is regular as well.*

The above result on regular configurations provides a clear characterization of the shape of the infection process. We now analyze the speed of the process. Observe that we can now assume that all configurations are regular (w.h.p.).

Lemma 3. *For any* $t < n$*, let* $\textsc{w}$ *be any white cell in* $\textsc{conf}\,(t+1)$ *and let* $\mathit{Red}(t)$ *be the set of red cells in* $\textsc{conf}\,(t)$*. It holds w.h.p. that*

$$d_{\square}(\textsc{w},\ \mathit{Red}(t+1)) \leqslant d_{\square}(\textsc{w},\ \mathit{Red}(t)) - 1$$

Starting from the initial configuration $\textsc{conf}\,(0)$ (that has one red cell), Lemma 3 implies that *every* white cell in $\mathcal{Q}$ w.h.p will become red within $O(D_{\square}(\mathcal{Q})) = O(L/R)$ time steps. Moreover, thanks to Lemmas 1 and 2, every red cell will become either red or black. Finally, when all cells are black or red, after the next time step there are no more white agent and the theorem follows.

4 Medium Transmission-Rate or Medium Mobility

We consider the network formed by n mobile agents with transmission radius $R \geqslant c_0\sqrt{\log n}$ and move radius ρ such that $R/2 \leq \rho \leq \alpha R^2/\sqrt{\log n}$ for sufficiently small constant $\alpha > 0$. We consider a grid partition of $\mathcal{Q}$ into cells of side length $\ell = R/(4\sqrt{2})$ The constant c_0 is chosen in order to guarantee the density condition (Eq. 1).

Theorem 3. *Under the above assumptions, the k-flooding time over $\mathcal{Q}$ is w.h.p. bounded by $\Theta(L/\rho)$.*

4.1 Proof of Theorem 3

We adopt Def.s 1, 2, and 3 given in the previous section. Moreover, we need the further

Definition 4. *Two cells are ρ-close if the (Euclidean) distance between their geometric centers is at most ρ.*

As in the previous section, we provide the local state-evolution law of any cell in a regular configuration.

Lemma 4. *Consider any cell $\textsc{c}$ at time step t and assume $\textsc{conf}\,(t)$ be regular. Then the following properties hold w.h.p.*
1) If $\textsc{c}$ is red-close then it becomes red in $\textsc{conf}\,(t+1)$.
2) If $\textsc{c}$ is ρ-close to a red-close cell, then $\textsc{c}$ becomes red in $\textsc{conf}\,(t+1)$.
3) If $\textsc{c}$ is white but it is not ρ-close to any red-close cell then it becomes white or red in $\textsc{conf}\,(t+1)$.
4) If $\textsc{c}$ is red or black but it is not ρ-close to any red-close cell then it becomes red or black in $\textsc{conf}\,(t+1)$.

As an easy consequence of the above lemma, we get the following

Lemma 5. *For any $t < n$, if $\textsc{conf}\,(t)$ is regular, then w.h.p. $\textsc{conf}\,(t+1)$ is regular as well.*

In what follows, we assume that all configurations are regular (w.h.p.) and analyze the speed of the infection process. Differently from the previous section, we will show this "speed" is $\Theta(\rho)$.

Lemma 6. *For any $t < n$, let $\textsc{w}$ be any white cell in $\textsc{conf}\,(t+1)$ and let Red-close$\,(t)$ be the set of red-close cells in $\textsc{conf}\,(t)$. It w.h.p. holds that*

$$d_{\square}(\textsc{w}, \textit{Red-close}\,(t+1)) \leqslant \max\{d_{\square}(\textsc{w}, \textit{Red-close}\,(t)) - \Theta(\rho/R), 0\}$$

The starting configuration $\textsc{conf}\,(0)$ is regular and contains one red cell. So, Lemma 6 implies that *every* white cell in $\mathcal{Q}$ w.h.p will become red within

$$O\left(\frac{D_{\square}(\mathcal{Q})\ell}{\rho}\right) = O\left(\frac{L}{\rho}\right) \text{ time steps}$$

Moreover, thanks to Lemma 4, every red cell will become either red or black. Finally, when all cells are black or red, in the next time step there are no more white agent and the theorem follows.

5 Low Transmission-Rate or High Mobility

In this section, we study the case when the move radius can be much larger than the transmission radius. More precisely, the move radius can be an arbitrary polynomial of the transmission radius, i.e. $\rho = O(\text{POLY}(R))$, we also assume $R \geqslant c_0\sqrt{\log n}$, for a sufficiently large c_0, and $\rho \geqslant 5R$.

A slightly different version of the geometric random-walk model is here adopted, the *cellular random walk*: the square $\mathcal{Q}$ is partitioned in squared *supercells* of edge length ρ. Then an agent lying in any position of a supercell C selects her next position independently and uniformly at random over the supercell neighborhood $N(C)$. Moreover, an agent can be informed by another (red) agent only if they both belong to the same supercell. So, in the cellular random walk model, the *influence* of an agent lying in a supercell C, in the next time step, is always restricted to the subregion $N(C)$. Observe that the cellular random walk model preserves all significant features of the standard one while, on the other hand, it avoids several geometric technicalities. The latter would yield a much more elaborate analysis without increasing the relevance of the obtained results.

We consider a cell partition of $\mathcal{Q}$ with $\ell = R/\sqrt{2}$ such that the grid of supercells is a subgrid of the grid of cells of side length ℓ. In what follows, the cells of size length ρ are called *supercells* and those of size length ℓ are called cells, simply. As usual, we assume $R \geqslant c_0 L\sqrt{\log n/n}$ for a sufficiently large constant c_0 that guarantees the density property w.r.t. the cells (and, consequently, w.r.t. the supercells).

Theorem 4. *Under the above assumptions and for ρ such that $5R \leqslant \rho \leqslant \text{POLY}(R)$, the k-flooding time over $\mathcal{Q}$ is w.h.p. $\Theta(L/\rho)$.*

5.1 Proof of Theorem 4

The higher agent mobility forces us to analyze the infection process over the supercells (besides over the cells). During the information spreading, the state a supercell can assume is defined by some bounds on the number of red and white agents inside it. Roughly speaking, the number of white agents must never be too small w.r.t. the number of red agents, moreover the latter must increase exponentially w.r.t. R^2. Since the infection process is rather complex, we need to consider a relative large number of possible supercell states. Our analysis will first show every supercell eventually evolves from the initial white state to the final black state according to a monotone process over a set of intermediate states.

Then, we will show that the speed of this process is asymptotically optimal, i.e., proportional to the move radius.

For a supercell C and a time step t, let $\#_r^t(C)$, $\#_w^t(C)$, and $\#_b^t(C)$ be, respectively, the number of red, white, and black agents in C. We define

$$\hat{h} \; = \; \left\lceil \log_{R^2}\left(c_0 \frac{\rho^2}{R^2} \log n\right)\right\rceil \tag{2}$$

Observe that the assumption $\rho = O(\textsc{poly}(R))$ implies $\hat{h} = \Theta(1)$.

Definition 5. *For any time t and for any supercell C, we define some possible states of C at time t.*

State $h = 0$ (White State): $\#_r^t(C) = 0$ AND $\#_b^t(C) = 0$.

State $h = 1, \ldots, \hat{h} - 1$ (Intermediate States): *The values of $\#_r^t(C)$ and $\#_w^t(C)$ satisfy*

$$a_h R^{2h} \leqslant \#_r^t(C) \leqslant b_h R^{2h} \quad \text{AND} \quad \#_w^t(C) \geqslant c_h \rho^2$$

where a_h, b_h, c_h are constants (suitably fixed - see the full version [10]) that satisfy $a_h < b_h$, $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_{\hat{h}-1} > 0$, $0 < b_1 \leqslant b_2 \leqslant \cdots \leqslant b_{\hat{h}-1}$, and $c_1 \geqslant c_2 \geqslant \cdots \geqslant c_{\hat{h}-1} > 0$.

State $\hat{h}$ (Red State): $\#_r^t(C) \geqslant 90\frac{\rho^2}{R^2}\log n$.

State $\hat{h}+1$ (Black State): $\#_w^t(C) = 0$.

All the above states are mutually disjoint but the last three ones, i.e., $\hat{h} - 1$, $\hat{h}$, $\hat{h}+1$. For every supercell C and time step t, in order to indicate that C satisfies the condition of state h, we will write $h^t(C) = h$.

Definition 6. *The configuration* $\textsc{conf}\,(t)$ *is regular if for every supercell C (1) an $h \in \{0, 1, \ldots, \hat{h}, \hat{h}+1\}$ exists s.t. $h^t(C) = h$ and (2) if $h^t(C) = \hat{h}+1$, it holds that,$\forall C' \in N(C)$, $h^t(C') = \hat{h} \vee h^t(C') = \hat{h}+1$.*

In the sequel, we exchange the order of the phases in a time step: the move phase now comes before the transmission one. Clearly, this does not change our asymptotical results. The next technical lemmas allow to control the 1-step evolution of the state of any supercell in terms of the number of red and white agents and how such agents spread over its cells. Remind that constants η_1, η_2 are defined in the density property (Eq. 1).

Lemma 7. *Let C be a supercell such that $\#_w^t(N(C)) \geqslant \lambda\rho^2$, for some constant $\lambda \geqslant 720/c_0^2$. Then, immediately after the move phase of time $t+1$ (and before the transmission phase), w.h.p., for every cell* c *in C, it holds that the number of white agents in* c *is at least $(\lambda/36)R^2$.*

Lemma 8. *Let C be a supercell such that $\#_r^t(C) \geqslant \lambda R^k$, for some constants $\lambda \geqslant 1800/c_0^2$ and $k \geqslant 2$. Then, immediately after the move phase of time $t+1$, w.h.p. in every supercell $C' \in N(C)$, the cells in C' hit by some red agent are at least* $\min\{(\lambda/30)R^k, \rho^2/(2R^2)\}$.

Lemma 9. *Let C be any supercell such that $h^t(C) = \hat{h}$. Then, immediately after the move phase of time $t+1$, w.h.p. for every supercell $C' \in N(C)$ all the cells in C' are hit by some red agent.*

Lemma 10. *For any time step t and any supercell C, let $\widehat{\#_r^t(C)} = \max\{\#_r^t(C'), C' \in N(C)\}$. If, for some $M > 0$, it holds that $\widehat{\#_r^t(C)} \leqslant M$, then it holds that $\#_r^{t+1}(C) \leqslant 68\eta_2 M R^2$ w.h.p.*

We are now able to provide the local state-evolution law of any supercell in a regular configuration. Let us define $m^t(C) = \max\{h \mid \exists C' \in N(C) :\ h^t(C') = h\}$.

Lemma 11. *if* CONF (t) *is regular then the following implications hold w.h.p., for every supercell C:*
(a) $m^t(C) = 0 \Rightarrow h^{t+1}(C) = 0$
(b) $1 \leqslant m^t(C) \leqslant \hat{h} - 1 \Rightarrow h^{t+1}(C) = m^t(C) + 1$
(c) $m^t(C) = \hat{h} \wedge (h^t(C) = h$ *with* $h < \hat{h}) \Rightarrow h^{t+1}(C) = \hat{h}$
(d) $m^t(C) = \hat{h} \wedge h^t(C) = \hat{h} \Rightarrow h^{t+1}(C) = \hat{h} + 1$
(e) $m^t(C) = \hat{h} + 1 \Rightarrow h^{t+1}(C) = \hat{h} + 1$

As a consequence of the above lemma, we get

Lemma 12. *For any $t < n$, if* CONF (t) *is regular, then w.h.p.* CONF $(t+1)$ *is regular as well.*

Lemma 13. *With high probability the initial configuration* CONF (0) *is regular and a supercell C exists such that $h^0(C) = 1$.*

For any time t, let Red(t) be the set of supercells whose state at time t is at least 1. For any supercell C, denote by $d_\square^t(C)$ the distance w.r.t. supercells between C and Red(t). Clearly, if $C \in \text{Red}(t)$ then $d_\square^t(C) = 0$.

Lemma 14. *For any $t \leqslant n$, if* CONF (t) *is regular, then w.h.p., let for any supercell W such that $h^t(W) = 0$, it holds that*

$$d_\square^{t+1}(W) \leqslant d_\square^t(W) - 1$$

Theorem 4 is an easy consequence of Lemmas 12, 13 and 14.

6 The Multi-source Case and the Completion Threshold

The multi-source k-flooding. Consider the k-flooding process with n agents over the square $\mathcal{Q}$ whose starting configuration contains an arbitrary subset of source agents. Every source agent has the same message (infection) and, again, the goal is to bound the completion time. Let A be the set of positions of the source agents at starting time.

For any point $x \in \mathcal{Q}$, define

$$d(x, A) = \min\{d(x, a) \mid a \in A\}$$

$$\text{ECC}(A) = \max\{d(x, A) \mid x \in \mathcal{Q}\}$$

The parameter ECC(A) is the geometric eccentricity of A in $\mathcal{Q}$.

Theorem 5. *Under the same assumptions of the single-source case, for any choice of the source positions A, the k-flooding time is w.h.p.*
i) $\Theta(\textsc{ecc}(A)/R)$*, for any* $\rho \leqslant R/(2\sqrt{2})$*;*
ii) $\Theta(\textsc{ecc}(A)/\rho)$*, for any* $R/2 \leqslant \rho \leqslant \textsc{poly}(R)$*.*

Sketch of Proof. The crucial observation is that in all cases (i.e. those of Sect.s 3, 4, and 5), the obtained local state-evolution law of the cells works for *any* starting configuration provided it is a regular one. It is easy to verify that, for any choice of the source subset, the starting configuration is w.h.p. regular. Moreover, our analysis of the speed of the information spreading does not change at all: the initial distance between every white cell (supercell) and its closest red cell (red-close supercell) decreases w.h.p. by 1 at any time step. In the case of more source agents, such initial distance is bounded by $\textsc{ecc}(A)$. □

The Completion Threshold for R. Consider the k-flooding protocol on the geometric-MANET of n agents over the square $\mathcal{Q}$ of side length $\sqrt{n}$. Then the following negative result holds.

Theorem 6. *Let k be any fixed positive constant, R be such that $R < \gamma\sqrt{\log n}$ for a sufficiently small positive constant γ, and ρ be such that $0 \leqslant \rho \leqslant \sqrt{n}$. Then, w.h.p. the k-flooding protocol does not complete.*

The proof (see [10]) shows the existence of some agents that w.h.p. do not "meet" any other agent for the first k time steps provided that R and k satisfy the theorem's hypothesis.

7 Conclusions

The probabilistic analysis of information spreading in mobile/dynamic networks is a challenging issue that is the subject of several current research projects and some relevant advances have been obtained in the last five years by using approaches based on time/space discrete approximation of the evolving systems and discrete probability [3,5,6,11,13,17,18]. We believe this *discrete* approach is rather promising to address several important related questions which are still far to be solved.

The more related open question to our work is the analysis of the k-flooding for non-constant k under the connectivity threshold. For instance, is the $\Theta(\log n)$-flooding able to complete significantly *under* the above threshold? Which is the role of the move radius ρ?

A more general challenging issue is to extend the analysis of the parsimonious flooding to other explicit models of MANET such as the random way-point model [15,9].

References

1. Ball, F., Neal, P.: A general model for stochastic SIR epidemics with two levels of mixing. Math. Biosci. 180, 73–102 (2002)
2. Ball, F., Neal, P.: Network epidemic models with two levels of mixing. Math. Biosci. 212, 69–87 (2008)

3. Baumann, H., Crescenzi, P., Fraigniaud, P.: Parsimonious flooding in dynamic graphs. In: Proc. of the 28th ACM PODC 2009 (2009)
4. Brauer, N., van den Driessche, P., Wu, J. (eds.): Mathematical Epidemiology. Lecture Notes in Mathematics, subseries in Mathematical Biosciences, 1945 (2008)
5. Clementi, A., Monti, A., Pasquale, F., Silvestri, R.: Broadcasting in dynamic radio networks. J. Comput. Syst. Sci. 75(4) (2009); (preliminary version in ACM PODC 2007)
6. Clementi, A., Macci, C., Monti, A., Pasquale, F., Silvestri, R.: Flooding time in edge-Markovian dynamic graphs. In: Proc of the 27th ACM PODC 2008 (2008)
7. Clementi, A., Monti, A., Pasquale, F., Silvestri, R.: Information spreading in stationary markovian evolving graphs. In: Proc. of the 23rd IEEE IPDPS 2009 (2009)
8. Clementi, A., Pasquale, F., Silvestri, R.: Manets: High mobility can make up for low transmission power. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 387–398. Springer, Heidelberg (2009)
9. Clementi, A., Monti, A., Silvestri, R.: Flooding over Manhattan. In: Proc of the 29th ACM PODC 2010 (2010)
10. Clementi, A., Silvestri, R.: Parsimonious Flooding in Geometric Random-Walks, ArXiv e-prints, 2011arXiv1101.5308C (2011)
11. Diaz, J., Mitsche, D., Perez-Gimenez, X.: On the connectivity of dynamic random geometric graphs. In: Proc. of 19th Annual ACM-SIAM SODA 2008, pp. 601–610 (2008)
12. Grossglauser, M., Tse, N.C.: Mobility increases the capacity of ad-hoc wireless networks. IEEE/ACM Trans. on Networking 10(4) (2002)
13. Jacquet, P., Mans, B., Rodolakis, G.: Information propagation speed in mobile and delay tolerant networks. In: Proc. of 29th IEEE INFOCOM 2009, pp. 244–252. IEEE, Los Alamitos (2009)
14. Kong, Z., Yeh, E.M.: On the latency for information dissemination in mobile wireless networks. In: Proc. of 9th ACM MobiHoc 2008, pp. 139–148 (2008)
15. Le Boudec, J.Y., Vojnovic, M.: The random trip model: stability, stationary regime, and perfect simulation. IEEE/ACM Trans. Netw. 14(6), 1153–1166 (2006)
16. McDiarmid, C.: On the method of bounded differences. In: Siemons, J. (ed.) London Mathematical Society Lecture Note, vol. 141, pp. 148–188. Cambridge University Press, Cambridge (1989)
17. Peres, Y., Sinclair, A., Sousi, P., Stauffer, A.: Mobile Geometric Graphs: Detection, Coverage and Percolation Eprint arXiv:1008.0075v2) (to appear in ACM SODA 2011) (2010)
18. Pettarin, A., Pietracaprina, A., Pucci, G., Upfal, E.: Infectious Random Walks. Eprint arXiv:1007.1604 (2010), Extended abstract will appear in (ACM PODC 2011)

Misleading Stars: What Cannot Be Measured in the Internet?[⋆]

Yvonne Anne Pignolet[1], Stefan Schmid[2], and Gilles Tredan[2]

[1] ABB Research, Switzerland
yvonne-anne.pignolet@ch.abb.com
[2] TU Berlin & T-Labs, Germany
{stefan,gilles}@net.t-labs.tu-berlin.de

Abstract. Traceroute measurements are one of the main instruments to shed light onto the structure and properties of today's complex networks such as the Internet. This paper studies the feasibility and infeasibility of inferring the network topology given traceroute data from a worst-case perspective, i.e., without any probabilistic assumptions on, e.g., the nodes' degree distribution. We attend to a scenario where some of the routers are anonymous, and propose two fundamental axioms that model two basic assumptions on the traceroute data: (1) each trace corresponds to a real path in the network, and (2) the routing paths are at most a factor $1/\alpha$ off the shortest paths, for some parameter $\alpha \in (0, 1]$. In contrast to existing literature that focuses on the cardinality of the set of (often only minimal) inferrable topologies, we argue that a large number of possible topologies alone is often unproblematic, as long as the networks have a similar structure. We hence seek to characterize the set of topologies inferred with our axioms. We introduce the notion of star graphs whose colorings capture the differences among inferred topologies; it also allows us to construct inferred topologies explicitly. We find that in general, inferrable topologies can differ significantly in many important aspects, such as the nodes' distances or the number of triangles. These negative results are complemented by a discussion of a scenario where the trace set is best possible, i.e., "complete". It turns out that while some properties such as the node degrees are still hard to measure, a complete trace set can help to determine global properties such as the connectivity.

1 Introduction

Surprisingly little is known about the structure of many important complex networks such as the Internet. One reason is the inherent difficulty of performing accurate, large-scale and preferably synchronous measurements from a large number of different vantage points. Another reason are privacy and information hiding issues: for example, network providers may seek to hide the details of their infrastructure to avoid tailored attacks.

Since knowledge of the network characteristics is crucial for many applications (e.g., *RMTP* [13], or *PaDIS* [14]), the research community implements measurement tools to

[⋆] Due to space constraints, some proofs are omitted in this document. They are available from the ArXiv document server (ID: 1105.5236).

analyze at least the main properties of the network. The results can then, e.g., be used to design more efficient network protocols in the future.

This paper focuses on the most basic characteristic of the network: its *topology*. The classic tool to study topological properties is *traceroute*. Traceroute allows us to collect traces from a given source node to a set of specified destination nodes. A trace between two nodes contains a sequence of identifiers describing a route between source and destination. However, not every node along such a path is configured to answer with its identifier. Rather, some nodes may be *anonymous* in the sense that they appear as stars ('$*$') in a trace. Anonymous nodes exacerbate the exploration of a topology because already a small number of anonymous nodes may increase the spectrum of inferrable topologies that correspond to a trace set $\mathcal{T}$.

This paper is motivated by the observation that the mere number of inferrable topologies alone does not contradict the usefulness or feasibility of topology inference; if the set of inferrable topologies is homogeneous in the sense that the different topologies share many important properties, the generation of all possible graphs can be avoided: an arbitrary representative may characterize the underlying network accurately. Therefore, we identify important topological metrics such as diameter or maximal node degree and examine how "close" the possible inferred topologies are with respect to these metrics.

1.1 Related Work

Arguably one of the most influential measurement studies on the Internet topology was conducted by the Faloutsos brothers [9] who show that the Internet exhibits a skewed structure: the nodes' out-degree follows a power-law distribution. Moreover, this property seems to be invariant over time. These results complement discoveries of similar distributions of communication traffic which is often self-similar, and of the topologies of natural networks such as human respiratory systems. This property allows us to give good predictions not only on node degree distributions but also, e.g., on the expected number of nodes at a given hop-distance. Since [9] was published, many additional results have been obtained, e.g., by conducting a distributed computing approach to increase the number of measurement points [7]. However, our understanding remains preliminary, and the topic continues to attract much attention from the scientific communities. In contrast to these measurement studies, we pursue a more formal approach, and a complete review of the empirical results obtained over the last years is beyond the scope of this paper.

In the field of *network tomography*, topologies are explored using pairwise end-to-end measurements, without the cooperation of nodes along these paths. This approach is quite flexible and applicable in various contexts, e.g., in social networks [5]. For a good discussion of this approach as well as results for a routing model along shortest and second shortest paths see [5]. For example, [5] shows that for sparse random graphs, a relatively small number of cooperating participants is sufficient to discover a network fairly well.

The classic tool to discover Internet topologies is traceroute [8]. Unfortunately, there are several problems with this approach that render topology inference difficult, such as *aliasing* or *load-balancing*, which has motivated researchers to develop new tools such as *Paris Traceroute* [6,11]. Another complication stems from the fact that routers may appear as stars in the trace since they are overloaded or since they are configured not to send out any ICMP responses. The lack of complete information in the trace set renders the accurate characterization of Internet topologies difficult.

This paper attends to the problem of anonymous nodes and assumes a conservative, "worst-case" perspective that does not rely on any assumptions on the underlying network. There are already several works on the subject. Yao et al. [16] initiated the study of possible candidate topologies for a given trace set and suggested computing the *minimal topology*, that is, the topology with the minimal number of anonymous nodes, which turns out to be NP-hard. Consequently, different heuristics have been proposed [10,11].

Our work is motivated by a series of papers by Acharya and Gouda. In [3], a network tracing theory model is introduced where nodes are "irregular" in the sense that each node appears in at least one trace with its real identifier. In [1], hardness results are derived for this model. However, as pointed out by the authors themselves, the irregular node model—where nodes are anonymous due to high loads—is less relevant in practice and hence they consider strictly anonymous nodes in their follow-up studies [2]. As proved in [2], the problem is still hard (in the sense that there are many minimal networks corresponding to a trace set), even with only two anonymous nodes, symmetric routing and without aliasing.

In contrast to this line of research on cardinalities, we are interested in the *network properties*. If the inferred topologies share the most important characteristics, the negative results in [1,2] may be of little concern. Moreover, we believe that a study limited to minimal topologies only may miss important redundancy aspects of the Internet. Unlike [1,2], our work is constructive in the sense that algorithms can be derived to compute inferred topologies.

Finally, in a broader context, Alon et al. [4] recently initiated the study of the multi-agent exploration of *link weights* in known network topologies, and derived bounds on the number of rounds and the number of agents required to complete the discovery of the edge weights or a shortest path.

1.2 Our Contribution

This paper initiates the study and characterization of topologies that can be inferred from a given trace set computed with the traceroute tool. While existing literature assuming a worst-case perspective has mainly focused on the cardinality of minimal topologies, we go one step further and examine specific topological graph properties.

We introduce a formal theory of topology inference by proposing basic axioms (i.e., assumptions on the trace set) that are used to guide the inference process. We present a novel definition for the isomorphism of inferred topologies which is aware of traffic paths; it is motivated by the observation that although two topologies look equivalent up to a renaming of anonymous nodes, the same trace set may result in different paths. Moreover, we initiate the study of two extremes: in the first scenario, we only require that each link appears at least once in the trace set; interestingly, however, it turns out that this is often not sufficient, and we propose a "best case" scenario where the trace set is, in some sense, *complete*: it contains paths between all pairs of non-anonymous nodes.

The main result of the paper is a negative one. It is shown that already a small number of anonymous nodes in the network renders topology inference difficult. In particular, we prove that in general, the possible inferrable topologies differ in many crucial aspects, e.g., the maximal node degree, the diameter, the stretch, the number of triangles and the number of connected components.

We introduce the concept of the *star graph* of a trace set that is useful for the characterization of inferred topologies. In particular, colorings of the star graphs allow us to constructively derive inferred topologies. (Although the general problem of computing the set of inferrable topologies is related to NP-hard problems such as *minimal graph coloring* and *graph isomorphism*, some important instances of inferrable topologies can be computed efficiently.) The chromatic number (i.e., the number of colors in the minimal proper coloring) of the star graph defines a lower bound on the number of anonymous nodes from which the stars in the traces could originate from. And the number of possible colorings of the star graph—a function of the *chromatic polynomial* of the star graph—gives an upper bound on the number of inferrable topologies. We show that this bound is tight in the sense that there are situations where there indeed exist so many inferrable topologies. Especially, there are problem instances where the cardinality of the set of inferrable topologies equals the *Bell number*. This insight complements (and generalizes to arbitrary, not only minimal, inferrable topologies) existing cardinality results.

Finally, we examine the scenario of *fully explored networks* for which "complete" trace sets are available. As expected, inferrable topologies are more homogenous and can be characterized well with respect to many properties such as node distances. However, we also find that other properties are inherently difficult to estimate. Interestingly, our results indicate that full exploration is often useful for global properties (such as connectivity) while it does not help much for more local properties (such as node degree).

2 Model

Let $\mathcal{T}$ denote the set of traces obtained from probing (e.g., by traceroute) a (not necessarily connected and undirected) network $G_0 = (V_0, E_0)$ with *nodes* or *vertices* V_0 (the set of routers) and *links* or *edges* E_0. We assume that G_0 is static during the probing time (or that probing is instantaneous). Each trace $T(u, v) \in \mathcal{T}$ describes a path connecting two nodes $u, v \in V_0$; when u and v do not matter or are clear from the context, we simply write T. Moreover, let $d_T(u, v)$ denote the distance (number of hops) between two nodes u and v in trace T. We define $d_{G_0}(u, v)$ to be the corresponding shortest path distance in G_0. Note that a trace between two nodes u and v may not describe the shortest path between u and v in G_0.

The nodes in V_0 fall into two categories: *anonymous* nodes and *non-anonymous* (or shorter: *named*) nodes. Therefore, each trace $T \in \mathcal{T}$ describes a sequence of symbols representing anonymous and non-anonymous nodes. We make the natural assumption that the first and the last node in each trace T is non-anonymous. Moreover, we assume that traces are given in a form where non-anonymous nodes appear with a unique, anti-aliased identifier (i.e., the multiple IP addresses corresponding to different interfaces of a node are resolved to one identifier); an anonymous node is represented as $*$ ("star") in the traces. For our formal analysis, we assign to each star in a trace set $\mathcal{T}$ a unique identifier i: $*_i$. (Note that except for the numbering of the stars, we allow identical copies of T in $\mathcal{T}$, and we do not make any assumptions on the implications of identical traces: they may or may not describe the same paths.) Thus, a trace $T \in \mathcal{T}$ is a sequence of symbols taken from an alphabet $\Sigma = \mathcal{ID} \cup (\bigcup_i *_i)$, where $\mathcal{ID}$ is the set of non-anonymous node identifiers (IDs): Σ is the union of the (anti-aliased) non-anonymous nodes and the set of all stars (with their unique identifiers) appearing in a trace set. The main challenge in topology inference is to determine which stars in the traces may originate from which anonymous nodes.

Henceforth, let $n = |\mathcal{ID}|$ denote the number of non-anonymous nodes and let $s = |\bigcup_i *_i|$ be the number of stars in $\mathcal{T}$; similarly, let a denote the number of anonymous nodes in a topology. Let $N = n + s = |\Sigma|$ be the total number of symbols occurring in $\mathcal{T}$.

Clearly, the process of topology inference depends on the assumptions on the measurements. In the following, we postulate the fundamental axioms that guide the reconstruction. First, we make the assumption that each link of G_0 is visited by the measurement process, i.e., it appears as a transition in the trace set $\mathcal{T}$. In other words, we are only interested in inferring the (sub-)graph for which measurement data is available.

AXIOM 0 (*Complete Cover*): Each edge of G_0 appears at least once in some trace in $\mathcal{T}$.

The next fundamental axiom assumes that traces always represent paths on G_0.

AXIOM 1 (*Reality Sampling*): For every trace $T \in \mathcal{T}$, if the distance between two symbols $\sigma_1, \sigma_2 \in T$ is $d_T(\sigma_1, \sigma_2) = k$, then there exists a path (i.e., a walk without cycles) of length k connecting two (named or anonymous) nodes σ_1 and σ_2 in G_0.

The following axiom captures the consistency of the routing protocol on which the traceroute probing relies. In the current Internet, policy routing is known to have in impact both on the route length [15] and on the convergence time [12].

AXIOM 2 (*α-(Routing) Consistency*): There exists an $\alpha \in (0, 1]$ such that, for every trace $T \in \mathcal{T}$, if $d_T(\sigma_1, \sigma_2) = k$ for two entries σ_1, σ_2 in trace T, then the shortest path connecting the two (named or anonymous) nodes corresponding to σ_1 and σ_2 in G_0 has distance at least $\lceil \alpha k \rceil$.

Note that if $\alpha = 1$, the routing is a shortest path routing. Moreover, note that if $\alpha = 0$, there can be loops in the paths, and there are hardly any topological constraints, rendering almost any topology inferrable. (For example, the complete graph with one anonymous router is always a solution.)

A natural axiom to merge traces is the following.

AXIOM 3 (*Trace Merging*): For two traces $T_1, T_2 \in \mathcal{T}$ for which $\exists \sigma_1, \sigma_2, \sigma_3$, where σ_2 refers to a named node, such that $d_{T_1}(\sigma_1, \sigma_2) = i$ and $d_{T_2}(\sigma_2, \sigma_3) = j$, it holds that the distance between two nodes u and v corresponding to σ_1 and σ_2, respectively, in G_0, is at most $d_{G_0}(\sigma_1, \sigma_3) \leq i + j$.

Any topology G which is consistent with these axioms (when applied to $\mathcal{T}$) is called *inferrable* from $\mathcal{T}$.

Definition 1 (Inferrable Topologies). *A topology G is (α-consistently)* inferrable *from a trace set $\mathcal{T}$ if axioms* AXIOM *0,* AXIOM *1,* AXIOM *2 (with parameter α), and* AXIOM *3 are fulfilled.*

We will refer by $\mathcal{G}_{\mathcal{T}}$ to the set of topologies inferrable from $\mathcal{T}$. Please note the following important observation.

Remark 1. In the absence of anonymous nodes, it holds that $G_0 \in \mathcal{G}_{\mathcal{T}}$, since $\mathcal{T}$ was generated from G_0 and AXIOM 0, AXIOM 1, AXIOM 2 and AXIOM 3 are fulfilled by definition. However, there are instances where an α-consistent trace set for G_0 contradicts AXIOM 0: as trace needs to start and end with a named node, some edges cannot appear in an α-consistent trace set $\mathcal{T}$. In the remainder of this paper, we will only consider settings where $G_0 \in \mathcal{G}_{\mathcal{T}}$.

The main objective of a topology inference algorithm ALG is to compute topologies which are consistent with these axioms. Concretely, ALG's input is the trace set $\mathcal{T}$ together with the parameter α specifying the assumed routing consistency. Essentially, the goal of any topology inference algorithm ALG is to compute a mapping of the symbols Σ (appearing in $\mathcal{T}$) to nodes in an inferred topology G; or, in case the input parameters α and $\mathcal{T}$ are contradictory, reject the input. This mapping of symbols to nodes implicitly describes the edge set of G as well: the edge set is unique as all the transitions of the traces in $\mathcal{T}$ are now unambiguously tied to two nodes.

So far, we have ignored an important and non-trivial question: When are two topologies $G_1, G_2 \in \mathcal{G}_{\mathcal{T}}$ different (and hence appear as two independent topologies in $\mathcal{G}_{\mathcal{T}}$)? In this paper, we pursue the following approach: We are not interested in purely topological isomorphisms, but we care about the identifiers of the non-anonymous nodes, i.e., we are interested in the locations of the non-anonymous nodes and their distance to other nodes. For anonymous nodes, the situation is slightly more complicated: one might think that as the nodes are anonymous, their "names" do not matter. Consider however the example in Figure 1: the two inferrable topologies have two anonymous nodes, one where $\{*_1, *_2\}$ plus $\{*_3, *_4\}$ are merged into one node each in the inferrable topology and one where $\{*_1, *_4\}$ plus $\{*_2, *_3\}$ are merged into one node each in the inferrable topology. In this paper, we regard the two topologies as different, for the following reason: Assume that there are two paths in the network, one $u \rightsquigarrow *_2 \rightsquigarrow v$ (e.g., during day time) and one $u \rightsquigarrow *_3 \rightsquigarrow v$ (e.g., at night); clearly, this traffic has different consequences and hence we want to be able to distinguish

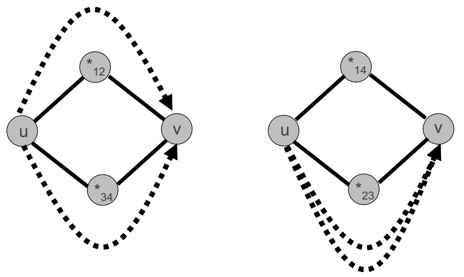


Fig. 1. Two non-isomorphic inferred topologies, i.e., different mapping functions lead to these topologies

between the two topologies described above. In other words, our notion of isomorphism of inferred topologies is *path-aware*.

It is convenient to introduce the following MAP function. Essentially, an inference algorithm computes such a mapping.

Definition 2 (Mapping Function MAP). *Let $G = (V, E) \in \mathcal{G_T}$ be a topology inferrable from $\mathcal{T}$. A topology inference algorithm describes a surjective mapping function* MAP $: \Sigma \to V$. *For the set of non-anonymous nodes in Σ, the mapping function is bijective; and each star is mapped to exactly one node in V, but multiple stars may be assigned to the same node. Note that for any $\sigma \in \Sigma$,* MAP(σ) *uniquely identifies a node $v \in V$. More specifically, we assume that* MAP *assigns labels to the nodes in V: in case of a named node, the label is simply the node's identifier; in case of anonymous nodes, the label is $*_\beta$, where β is the concatenation of the* sorted *indices of the stars which are merged into node $*_\beta$.*

With this definition, two topologies $G_1, G_2 \in \mathcal{G_T}$ differ if and only if they do not describe the identical (MAP-) labeled topology. We will use this MAP function also for G_0, i.e., we will write MAP(σ) to refer to a symbol σ's corresponding node in G_0.

In the remainder of this paper, we will often assume that AXIOM 0 is given. Moreover, note that AXIOM 3 is redundant. Therefore, in our proofs, we will not explicitly cover AXIOM 0, and it is sufficient to show that AXIOM 1 holds to prove that AXIOM 3 is satisfied.

Lemma 1. AXIOM *1 implies* AXIOM *3.*

PROOF. Let $\mathcal{T}$ be a trace set, and $G \in \mathcal{G_T}$. Let $\sigma_1, \sigma_2, \sigma_3$ s.t. $\exists T_1, T_2 \in \mathcal{T}$ with $\sigma_1 \in T_1, \sigma_3 \in T_2$ and $\sigma_2 \in T_1 \cap T_2$. Let $i = d_{T_1}(\sigma_1, \sigma_2)$ and $j = d_{T_2}(\sigma_1, \sigma_3)$. Since any inferrable topology G fulfills AXIOM 1, there is a path π_1 of length at most i between the nodes corresponding to σ_1 and σ_2 in G and a path π_2 of length at most j between the nodes corresponding to σ_2 and σ_3 in G. The combined path can only be shorter, and hence the claim follows. □

3 Inferrable Topologies

What insights can be obtained from topology inference with minimal assumptions, i.e., with our axioms? Or what is the structure of the inferrable topology set $\mathcal{G_T}$? We first make some general observations and then examine different graph metrics in more detail.

3.1 Basic Observations

Although the generation of the entire topology set $\mathcal{G_T}$ may be computationally hard, some instances of $\mathcal{G_T}$ can be computed efficiently. The simplest possible inferrable topology is the so-called *canonic graph G_C*: the topology which assumes that all stars in the traces refer to different anonymous nodes. In other words, if a trace set $\mathcal{T}$ contains $n = |\mathcal{ID}|$ named nodes and s stars, G_C will contain $|V(G_C)| = N = n + s$ nodes.

Definition 3 (Canonic Graph G_C). *The* canonic graph *is defined by $G_C(V_C, E_C)$ where $V_C = \Sigma$ is the set of (anti-aliased) nodes appearing in $\mathcal{T}$ (where each star is considered a unique anonymous node) and where $\{\sigma_1, \sigma_2\} \in E_C \Leftrightarrow \exists T \in \mathcal{T}, T = (\ldots, \sigma_1, \sigma_2, \ldots)$, i.e., σ_1 follows after σ_2 in some trace T ($\sigma_1, \sigma_2 \in T$ can be either non-anonymous nodes or stars). Let $d_C(\sigma_1, \sigma_2)$ denote the* canonic distance *between two nodes, i.e., the length of a shortest path in G_C between the nodes σ_1 and σ_2.*

Note that G_C is indeed an inferrable topology. In this case, MAP $: \Sigma \rightarrow \Sigma$ is the identity function.

Theorem 1. *G_C is inferrable from $\mathcal{T}$.*

G_C can be computed efficiently from $\mathcal{T}$: represent each non-anonymous node and star as a separate node, and for any pair of consecutive entries (i.e., nodes) in a trace, add the corresponding link. The time complexity of this construction is linear in the size of $\mathcal{T}$.

With the definition of the canonic graph, we can derive the following lemma which establishes a necessary condition when two stars cannot represent the same node in G_0 from constraints on the routing paths. This is useful for the characterization of inferred topologies.

Lemma 2. *Let $*_1, *_2$ be two stars occurring in some traces in $\mathcal{T}$. $*_1, *_2$ cannot be mapped to the same node, i.e., $\text{MAP}(*_1) \neq \text{MAP}(*_2)$, without violating the axioms in the following conflict situations:*

(i) if $*_1 \in T_1$ and $*_2 \in T_2$, and T_1 describes too a long path between anonymous node $\text{MAP}(*_1)$ and non-anonymous node u, i.e., $\lceil \alpha \cdot d_{T_1}(*_1, u) \rceil > d_C(u, *_2)$.
(ii) if $*_1 \in T_1$ and $*_2 \in T_2$, and there exists a trace T that contains a path between two non-anonymous nodes u and v and $\lceil \alpha \cdot d_T(u, v) \rceil > d_C(u, *_1) + d_C(v, *_2)$.

PROOF. The first proof is by contradiction. Assume $\text{MAP}(*_1) = \text{MAP}(*_2)$ represents the same node v of G_0, and that $\lceil \alpha \cdot d_{T_1}(v, u) \rceil > d_C(u, v)$. Then we know from AXIOM 2 that $d_C(v, u) \geq d_{G_0}(v, u) \geq \lceil \alpha \cdot d_{T_1}(u, v) \rceil > d_C(v, u)$, which yields the desired contradiction.

Similarly for the second proof, assume for the sake of contradiction that $\text{MAP}(*_1) = \text{MAP}(*_2)$ represents the same node w of G_0, and that $\lceil \alpha \cdot d_T(u, v) \rceil > d_C(u, *_1) + d_C(v, *_2) \geq d_{G_0}(u, w) + d_{G_0}(v, w)$. Due to the triangle inequality, we have that $d_{G_0}(u, w) + d_{G_0}(v, w) \geq d_{G_0}(u, v)$ and hence, $\lceil \alpha \cdot d_T(u, v) \rceil > d_{G_0}(u, v)$, which contradicts the fact that G_0 is inferrable (Remark 1). □

Lemma 2 can be applied to show that a topology is not inferrable from a given trace set because it merges (i.e., maps to the same node) two stars in a manner that violates the axioms. Let us introduce a useful concept for our analysis: the *star graph* that describes the conflicts between stars.

Definition 4 (Star Graph G_*). *The* star graph *$G_*(V_*, E_*)$ consists of vertices V_* representing stars in traces, i.e., $V_* = \bigcup_i *_i$. Two vertices are connected if and only if they must differ according to Lemma 2, i.e., $\{*_1, *_2\} \in E_*$ if and only if at least one of the conditions of Lemma 2 hold for $*_1, *_2$.*

Note that the star graph G_* is unique and can be computed efficiently for a given trace set $\mathcal{T}$: Conditions (i) and (ii) can be checked by computing G_C. However, note that while G_* specifies some stars which cannot be merged, the construction is not sufficient: as Lemma 2 is based on G_C, additional links might be needed to characterize the set of inferrable and α-consistent topologies $\mathcal{G}_\mathcal{T}$ exactly. In other words, a topology G obtained by merging stars that are adjacent in G_* is never inferrable ($G \notin \mathcal{G}_\mathcal{T}$); however, merging non-adjacent stars does not guarantee that the resulting topology is inferrable.

What do star graphs look like? The answer is *arbitrarily*: the following lemma states that the set of possible star graphs is equivalent to the class of general graphs. This claim holds for any α.

Lemma 3. *For any graph $G = (V, E)$, there exists a trace set $\mathcal{T}$ such that G is the star graph for $\mathcal{T}$.*

The problem of computing inferrable topologies is related to the vertex colorings of the star graphs. We will use the following definition which relates a vertex coloring of G_* to an inferrable topology G by contracting independent stars in G_* to become one anonymous node in G. For example, observe that a maximum coloring treating every star in the trace as a separate anonymous node describes the inferrable topology G_C.

Definition 5 (Coloring-Induced Graph). *Let γ denote a coloring of G_* which assigns colors $1, \ldots, k$ to the vertices of G_*: $\gamma : V_* \rightarrow \{1, \ldots, k\}$. We require that γ is a proper coloring of G_*, i.e., that different anonymous nodes are assigned different colors: $\{u, v\} \in E_* \Rightarrow \gamma(u) \neq \gamma(v)$. G_γ is defined as the topology* induced *by γ. G_γ describes the graph G_C where nodes of the same color are contracted: two vertices u and v represent the same node in G_γ, i.e.,* $\text{MAP}(*_i) = \text{MAP}(*_j)$, *if and only if $\gamma(*_i) = \gamma(*_j)$.*

The following two lemmas establish an intriguing relationship between colorings of G_* and inferrable topologies. Also note that Definition 5 implies that two different colorings of G_* define two non-isomorphic inferrable topologies.

We first show that while a coloring-induced topology always fulfills AXIOM 1, the routing consistency is sacrificed.

Lemma 4. *Let γ be a proper coloring of G_*. The coloring induced topology G_γ is a topology fulfilling* AXIOM *2 with a routing consistency of α', for some positive α'.*

An inferrable topology always defines a proper coloring on G_*.

Lemma 5. *Let $\mathcal{T}$ be a trace set and G_* its corresponding star graph. If a topology G is inferrable from $\mathcal{T}$, then G induces a proper coloring on G_*.*

The colorings of G_* allow us to derive an upper bound on the cardinality of $\mathcal{G}_\mathcal{T}$.

Theorem 2. *Given a trace set $\mathcal{T}$ sampled from a network G_0 and $\mathcal{G}_\mathcal{T}$, the set of topologies inferrable from $\mathcal{T}$, it holds that:*

$$\sum_{k=\gamma(G_*)}^{|V_*|} P(G_*, k)/k! \geq |\mathcal{G}_\mathcal{T}|,$$

where $\gamma(G_*)$ is the chromatic number of G_* and $P(G_*, k)$ is the number of colorings of G_* with k colors (known as the chromatic polynomial *of G_*).*

PROOF. The proof follows directly from Lemma 5 which shows that each inferred topology has proper colorings, and the fact that a coloring of G_* cannot result in two different inferred topologies, as the coloring uniquely describes which stars to merge (Lemma 4). In order to account for isomorphic colorings, we need to divide by the number of color permutations. □

Note that the fact that G_* can be an arbitrary graph (Lemma 3) implies that we cannot exploit some special properties of G_* to compute colorings of G_* and $\gamma(G_*)$. Also note that the exact computation of the upper bound is hard, since the minimal coloring as well as the chromatic polynomial of G_* (in P♯) is needed. To complement the upper bound, we note that star graphs with a small number of conflict edges can indeed result in a large number of inferred topologies.

Theorem 3. *For any $\alpha > 0$, there is a trace set for which the number of non-isomorphic colorings of G_* equals $|\mathcal{G}_\mathcal{T}|$, in particular $|\mathcal{G}_\mathcal{T}| = B_s$, where $\mathcal{G}_\mathcal{T}$ is the set of inferrable and α-consistent topologies, s is the number of stars in $\mathcal{T}$, and B_s is the* Bell number *of s. Such a trace set can originate from a G_0 network with one anonymous node only.*

PROOF. Consider a trace set $\mathcal{T} = \{(\sigma_i, *_i, \sigma_i')_{i=1,\dots,s}\}$ (e.g., obtained from exploring a topology G_0 where one anonymous center node is connected to $2s$ named nodes). The trace set does not impose any constraints on how the stars relate to each other, and hence, G_* does not contain any edges at all; even when stars are merged, there are no constraints on how the stars relate to each other. Therefore, the star graph for $\mathcal{T}$ has $B_s = \sum_{j=0}^{s} S_{(s,j)}$ colorings, where $S_{(s,j)} = 1/j! \cdot \sum_{\ell=0}^{j} (-1)^\ell \binom{j}{\ell} (j-\ell)^s$ is the number of ways to group s nodes into j different, disjoint non-empty subsets (known as the *Stirling number of the second kind*). Each of these colorings also describes a distinct inferrable topology as MAP assigns unique labels to anonymous nodes stemming from merging a group of stars (cf Definition 2). □

3.2 Properties

Even if the number of inferrable topologies is large, topology inference can still be useful if one is mainly interested in the properties of G_0 and if the ensemble $\mathcal{G}_\mathcal{T}$ is homogenous with respect to these properties; for example, if "most" of the instances in $\mathcal{G}_\mathcal{T}$ are close to G_0, there may be an option to conduct an efficient sampling analysis on random representatives. Therefore, in the following, we will take a closer look how much the members of $\mathcal{G}_\mathcal{T}$ differ.

Important metrics to characterize inferrable topologies are, for instance, the graph size, the diameter DIAM$(\cdot)$, the number of triangles $C_3(\cdot)$ of G, and so on. In the following, let $G_1 = (V_1, E_1), G_2 = (V_2, E_2) \in \mathcal{G}_\mathcal{T}$ be two arbitrary representatives of $\mathcal{G}_\mathcal{T}$.

As one might expect, the graph size can be estimated quite well.

Lemma 6. *It holds that $|V_1| - |V_2| \leq s - \gamma(G_*) \leq s - 1$ and $|V_1|/|V_2| \leq (n+s)/(n+\gamma(G_*)) \leq (2+s)/3$. Moreover, $|E_1| - |E_2| \leq 2(s - \gamma(G_*))$ and $|E_1|/|E_2| \leq (\nu + 2s)/(\nu + 2) \leq s$, where ν denotes the number of edges between non-anonymous nodes. There are traces with inferrable topology G_1, G_2 reaching these bounds.*

Observe that inferrable topologies can also differ in the number of connected components. This implies that the shortest distance between two named nodes can differ arbitrarily between two representatives in $\mathcal{G}_\mathcal{T}$.

Lemma 7. *Let* $\textsc{Comp}(G)$ *denote the number of connected components of a topology* G. *Then,* $|\textsc{Comp}(G_1) - \textsc{Comp}(G_2)| \le n/2$. *There are traces with inferrable topology* G_1, G_2 *reaching these bounds.*

An important criterion for topology inference regards the distortion of shortest paths.

Definition 6 (Stretch). *The maximal ratio of the distance of two non-anonymous nodes in* G_0 *and a connected topology* G *is called the* stretch ρ: $\rho = \max_{u,v \in \mathcal{ID}(G_0)} \max\{d_{G_0}(u,v)/d_G(u,v), d_G(u,v)/d_{G_0}(u,v)\}$.

From Lemma 7 we already know that inferrable topologies can differ in the number of connected components, and hence, the distance and the stretch between nodes can be arbitrarily wrong. Hence, in the following, we will focus on connected graphs only. However, even if two nodes are connected, their distance can be much longer or shorter than in G_0.

Figure 2 gives an example. Both topologies are inferrable from the traces $T_1 = (v, *, v_1, \ldots, v_k, u)$ and $T_2 = (w, *, w_1, \ldots, w_k, u)$. One inferrable topology is the canonic graph G_C (Figure 2 *left*), whereas the other topology merges the two anonymous nodes (Figure 2 *right*). The distances between v and w are $2(k+2)$ and 2, respectively, implying a stretch of $k+2$.

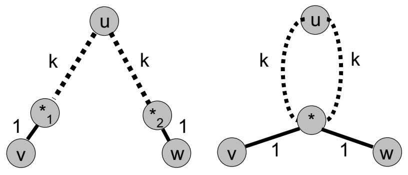


Fig. 2. Due to the lack of a trace between v and w, the stretch of an inferred topology can be large

Lemma 8. *Let* u *and* v *be two arbitrary named nodes in the connected topologies* G_1 *and* G_2. *Then, even for only two stars in the trace set, it holds for the stretch that* $\rho \le (N-1)/2$. *There are traces with inferrable topology* G_1, G_2 *reaching these bounds.*

We now turn our attention to the diameter and the degree.

Lemma 9. *For connected topologies* G_1, G_2 *it holds that* $\textsc{Diam}(G_1) - \textsc{Diam}(G_2) \le (s-1)/s \cdot \textsc{Diam}(G_C) \le (s-1)(N-1)/s$ *and* $\textsc{Diam}(G_1)/\textsc{Diam}(G_2) \le s$, *where* $\textsc{Diam}$ *denotes the graph diameter and* $\textsc{Diam}(G_1) > \textsc{Diam}(G_2)$. *There are instances* G_1, G_2 *that reach these bounds.*

Proof. *Upper bound:* As G_C does not merge any stars, it describes the network with the largest diameter. Let π be a longest path between two nodes u and v in G_C. In the extreme case, π is the only path determining the network diameter and π contains all star nodes. Then, the graph where all s stars are merged into one anonymous node has a minimal diameter of at least $\textsc{Diam}(G_C)/s$.

Example which meets the bound: Consider the trace set $\mathcal{T} = \{(u_1, \ldots, *_1, \ldots, u_2), (u_2, \ldots, *_2, \ldots, u_3), \ldots, (u_s, \ldots, *_s, \ldots, u_{s+1})\}$ with x named nodes and star in the middle between u_i and u_{i+1} (assume x to be even, x does not include u_i and u_{i+1}). It holds that $\textsc{Diam}(G_C) = s \cdot (x+2)$ whereas in a graph G where all stars are merged, $\textsc{Diam}(G) = x + 2$. There are $n = s(x+1)$ non-anonymous nodes, so $x = (n - s - 1)/s$. Figure 3 depicts an example. □

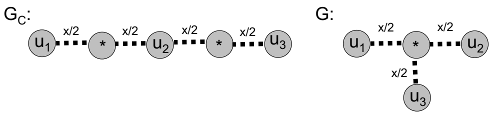


Fig. 3. Estimation error for diameter

Lemma 10. *For the maximal node degree* $\textsc{Deg}$*, we have* $\textsc{Deg}(G_1) - \textsc{Deg}(G_2) \leq 2(s - \gamma(G_*))$ *and* $\textsc{Deg}(G_1)/\textsc{Deg}(G_2) \leq s - \gamma(G_*) + 1$*. There are instances* G_1, G_2 *that reach these bounds.*

Another important topology measure that indicates how well meshed the network is, is the number of triangles.

Lemma 11. *Let* $C_3(G)$ *be the number of cycles of length* 3 *of the graph* G*. It holds that* $C_3(G_1) - C_3(G_2) \leq 2s(s-1)$*, which can be reached. The relative error* $C_3(G_1)/C_3(G_2)$ *can be arbitrarily large unless the number of links between non-anonymous nodes exceeds* $n^2/4$ *in which case the ratio is upper bounded by* $2s(s-1)+1$*.*

4 Full Exploration

So far, we assumed that the trace set $\mathcal{T}$ contains each node and link of G_0 at least once. At first sight, this seems to be the best we can hope for. However, sometimes traces exploring the vicinity of anonymous nodes in different ways yields additional information that help to characterize $\mathcal{G}_{\mathcal{T}}$ better.

This section introduces the concept of *fully explored networks*: $\mathcal{T}$ contains sufficiently many traces such that the distances between non-anonymous nodes can be estimated accurately.

Definition 7 (Fully Explored Topologies). *A topology* G_0 *is fully explored by a trace set* $\mathcal{T}$ *if it contains all nodes and links of* G_0 *and for each pair* $\{u, v\}$ *of non-anonymous nodes in the same component of* G_0 *there exists a trace* $T \in \mathcal{T}$ *containing both nodes* $u \in T$ *and* $v \in T$*.*

In some sense, a trace set for a fully explored network is the best we can hope for. Properties that cannot be inferred well under the fully explored topology model are infeasible to infer without additional assumptions on G_0. In this sense, this section provides upper bounds on what can be learned from topology inference, and accordingly, we will constrain ourselves to routing along shortest paths only ($\alpha = 1$).

Let us again study the properties of the family of inferrable topologies fully explored by a trace set. Obviously, all the upper bounds from Section 3 are still valid for fully explored topologies. In the following, let $G_1, G_2 \in \mathcal{G}_{\mathcal{T}}$ be arbitrary representatives of

$\mathcal{G}_\mathcal{T}$ for a fully explored trace set $\mathcal{T}$. A direct consequence of the Definition 7 concerns the number of connected components and the stretch. (Recall that the stretch is defined with respect to named nodes only, and since $\alpha = 1$, a 1-consistent inferrable topology cannot include a shorter path between u and v than the one that must appear in a trace of $\mathcal{T}$.)

Lemma 12. *It holds that* $\text{COMP}(G_1) = \text{COMP}(G_2)$ *(=* $\text{COMP}(G_0)$*) and the stretch is 1.*

The proof for the claims of the following lemmata are analogous to our former proofs, as the main difference is the fact that there might be more conflicts, i.e., edges in G_*.

Lemma 13. *For fully explored networks it holds that* $|V_1| - |V_2| \leq s - \gamma(G_*) \leq s - 1$ *and* $|V_1|/|V_2| \leq (n+s)/(n+\gamma(G_*)) \leq (2+s)/3$. *Moreover,* $|E_1| - |E_2| \in 2(s-\gamma(G_*))$ *and* $|E_1|/|E_2| \leq (\nu + 2s)/(\nu + 2) \leq s$*, where* ν *denotes the number of links between non-anonymous nodes. There are traces with inferrable topology* G_1, G_2 *reaching these bounds.*

Lemma 14. *For the maximal node degree, we have* $\text{DEG}(G_1) - \text{DEG}(G_2) \leq 2(s - \gamma(G_*))$ *and* $\text{DEG}(G_1)/\text{DEG}(G_2) \leq s - \gamma(G_*) + 1$. *There are instances* G_1, G_2 *that reach these bounds.*

From Lemma 12 we know that fully explored scenarios yield a perfect stretch of one. However, regarding the diameter, the situation is different in the sense that distances between anonymous nodes play a role.

Lemma 15. *For connected topologies* G_1, G_2 *it holds that* $\text{DIAM}(G_1)/\text{DIAM}(G_2) \leq 2$*, where* DIAM *denotes the graph diameter and* $\text{DIAM}(G_1) > \text{DIAM}(G_2)$. *There are instances* G_1, G_2 *that reach this bound. Moreover, there are instances with* $\text{DIAM}(G_1) - \text{DIAM}(G_2) = s/2$.

The number of triangles with anonymous nodes can still not be estimated accurately in the fully explored scenario.

Lemma 16. *There exist graphs where* $C_3(G_1) - C_3(G_2) = s(s-1)/2$*, and the relative error* $C_3(G_1)/C_3(G_2)$ *can be arbitrarily large.*

Property/Scenario	Arbitrary		Fully Explored ($\alpha = 1$)	
	$G_1 - G_2$	G_1/G_2	$G_1 - G_2$	G_1/G_2
# of nodes	$\leq s - \gamma(G_*)$	$\leq (n+s)/(n+\gamma(G_*))$	$\leq s - \gamma(G_*)$	$\leq (n+s)/(n+\gamma(G_*))$
# of links	$\leq 2(s - \gamma(G_*))$	$\leq (\nu + 2s)/(\nu + 2)$	$\leq 2(s - \gamma(G_*))$	$\leq (\nu + 2s)/(\nu + 2)$
# of connected components	$\leq n/2$	$\leq n/2$	$= 0$	$= 1$
Stretch	-	$\leq (N-1)/2$	-	$= 1$
Diameter	$\leq (s-1)/s \cdot (N-1)$	$\leq s$	$s/2$ (¶)	2
Max. Deg.	$\leq 2(s - \gamma(G_*))$	$\leq s - \gamma(G_*) + 1$	$\leq 2(s - \gamma(G_*))$	$\leq s - \gamma(G_*) + 1$
Triangles	$\leq 2s(s-1)$	∞	$\leq 2s(s-1)/2$	∞

Fig. 4. Summary of our bounds on the properties of inferrable topologies. s denotes the number of stars in the traces, n is the number of named nodes, $N = n + s$, and ν denotes the number of links between named nodes. Note that trace sets meeting these bounds exist for all properties for which we have tight or upper bounds. For the entry marked with (¶), only "lower bounds" are derived, i.e., examples that yield at least the corresponding accuracy; as the upper bounds from the arbitrary scenario do not match, how to close the gap remains an open question.

5 Conclusion

We understand our work as a first step to shed light onto the similarity of inferrable topologies based on most basic axioms and without any assumptions on power-law properties, i.e., in the worst case. Using our formal framework we show that the topologies for a given trace set may differ significantly. Thus, it is impossible to accurately characterize topological properties of complex networks. To complement the general analysis, we propose the notion of fully explored networks or trace sets, as a "best possible scenario". As expected, we find that fully exploring traces allow us to determine several properties of the network more accurately; however, it also turns out that even in this scenario, other topological properties are inherently hard to compute. Our results are summarized in Figure 4.

Our work opens several directions for future research. So far we have only investigated fully explored networks with short path routing ($\alpha = 1$), and a scenario with suboptimal routes still needs to be investigated. One may also study whether the minimal inferrable topologies considered in, e.g., [1,2], are more similar in nature. More importantly, while this paper presented results for the general worst-case, it would be interesting to devise algorithms that compute, for a *given trace set*, worst-case bounds for the properties under consideration. For example, such approximate bounds would be helpful to decide whether additional measurements are needed. Moreover, maybe such algorithms may even give advice on the locations at which such measurements would be most useful.

Acknowledgments. Stefan Schmid would like to thank H. B. Acharya and Ingmar Poese for initial discussions in Calcutta and Berlin. We are also grateful to Steve Uhlig and Bernhard Ager for their comments and the literature pointers. A full version of this article including all proofs is available on ArXiv (ID: 1105.5236).

References

1. Acharya, H., Gouda, M.: The weak network tracing problem. In: Kant, K., Pemmaraju, S.V., Sivalingam, K.M., Wu, J. (eds.) ICDCN 2010. LNCS, vol. 5935, pp. 184–194. Springer, Heidelberg (2010)
2. Acharya, H., Gouda, M.: On the hardness of topology inference. In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 251–262. Springer, Heidelberg (2011)
3. Acharya, H.B., Gouda, M.G.: A theory of network tracing. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 62–74. Springer, Heidelberg (2009)
4. Alon, N., Emek, Y., Feldman, M., Tennenholtz, M.: Economical graph discovery. In: Proc. 2nd Symposium on Innovations in Computer Science, ICS (2011)
5. Anandkumar, A., Hassidim, A., Kelner, J.: Topology discovery of sparse random graphs with few participants. In: Proc. SIGMETRICS (2011)
6. Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., Friedman, T., Latapy, M., Magnien, C., Teixeira, R.: Avoiding traceroute anomalies with paris traceroute. In: Proc. 6th ACM SIGCOMM Conference on Internet Measurement (IMC), pp. 153–158 (2006)
7. Buchanan, M.: Data-bots chart the internet. Science 813(3) (2005)

8. Cheswick, B., Burch, H., Branigan, S.: Mapping and visualizing the internet. In: Proc. USENIX Annual Technical Conference, ATEC (2000)
9. Faloutsos, M., Faloutsos, P., Faloutsos, C.: On power-law relationships of the internet topology. In: Proc. SIGCOMM, pp. 251–262 (1999)
10. Gunes, M., Sarac, K.: Resolving anonymous routers in internet topology measurement studies. In: Proc. INFOCOM (2008)
11. Jin, X., Yiu, W.-P., Chan, S.-H., Wang, Y.: Network topology inference based on end-to-end measurements. IEEE Journal on Selected Areas in Communications 24(12), 2182–2195 (2006)
12. Labovitz, C., Ahuja, A., Venkatachary, S., Wattenhofer, R.: The impact of internet policy and topology on delayed routing convergence. In: Proc. 20th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM (2001)
13. Paul, S., Sabnani, K.K., Lin, J.C., Bhattacharyya, S.: Reliable multicast transport protocol (rmtp). IEEE Journal on Selected Areas in Communications 5(3) (1997)
14. Poese, I., Frank, B., Ager, B., Smaragdakis, G., Feldmann, A.: Improving content delivery using provider-aided distance information. In: Proc. ACM IMC (2010)
15. Tangmunarunkit, H., Govindan, R., Shenker, S., Estrin, D.: The impact of routing policy on internet paths. In: Proc. INFOCOM, vol. 2, pp. 736–742 (2002)
16. Yao, B., Viswanathan, R., Chang, F., Waddington, D.: Topology inference in the presence of anonymous routers. In: Proc. IEEE INFOCOM, pp. 353–363 (2003)

Brief Announcement: A Randomized Algorithm for Label Assignment in Dynamic Networks

Meg Walraed-Sullivan, Radhika Niranjan Mysore, Keith Marzullo, and Amin Vahdat

University of California, San Diego
La Jolla CA, USA
{mwalraed,radhika,marzullo,vahdat}@cs.ucsd.edu

In large-scale networking environments, such as data centers, a key difficulty is the assignment of labels to network elements. Labels can be assigned statically, e.g. MAC-addresses in traditional Layer 2 networks, or by a central authority as in DHCP in Layer 3 networks. On the other hand, networks requiring a dynamic solution often use a Consensus-based state machine approach. While designing Alias [2], a protocol for automatically assigning hierarchically meaningful addresses in data center networks, we encountered an instance of label assignment with entirely different requirements. In this case, the rules for labels depend on connectivity, and connectivity (and hence, labels) changes over time. Thus, neither static assignment nor a state machine approach is ideal.

In the context of large scale data center networks, practical constraints are important. A centralized solution introduces a single point of failure and necessitates either flooding or a separate out-of-band control network to establish communication between the centralized component and all network elements; this is problematic at the scale of a data center. We also require a solution that scales, is robust in the face of miswirings and transient startup conditions, and has low message overhead and convergence time. To this end, we specify the Label Selection Problem (LSP), which is the problem of practical label assignment in data center networks.

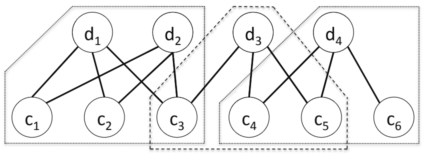


Fig. 1. *Sample Topology*

In LSP, we consider topologies made up of *chooser* processes connected to *decider* processes, as shown in Fig. 1. More formally, each chooser c has a set *c.deciders* of deciders associated with it. This set can change over time. Each chooser c is connected to each decider in *c.deciders* with a fair lossy link. Such links can drop messages, but if two processes p and q are connected by a fair lossy link and p sends m infinitely often to q, then q will receive m infinitely often. Both decider and chooser processes can *crash* in a failstop manner (thus going from *up* to *down*) and can *recover* (thus going from *down* to *up*) at any time. We assume that a process writes its state to stable storage before sending a set of messages. When a process recovers, it is restored to the state before sending the last set of message; duplicate messages may be sent upon recovery. So, we treat recovered processes as perhaps slow processes, and allow for duplicate messages.

We require an assignment of labels to choosers such that any two choosers that are connected to the same decider have distinct labels. Figure 1 illustrates sets of choosers and the deciders they share. For instance, chooser c_3 shares deciders d_1 and d_2 with choosers c_1 and c_2 and shares d_3 with c_4 and c_5. Because of this, c_3 may not select the same label as any of choosers c_1, c_2, c_4 and c_5, but c_3 and c_6 are free to select the same label. We denote c's current choice of label with $c.me$: $c.me = \perp$ indicates that c has not chosen a label.

We specify LSP with two properties, **Progress:** For each chooser c, once c remains up, eventually $c.me \neq \perp$ and **Distinctness:** For each distinct pair of choosers c_1 and c_2, once c_1 and c_2 remain up and there is some decider that remains up and remains in $c_1.deciders \cap c_2.deciders$, eventually always $c_1.me \neq c_2.me$. A key difficulty in solving LSP is that a chooser can not necessarily know when its choice satisfies **Distinctness**. This is because its set $c.deciders$ can change over time, thereby introducing new conflicts.

To solve LSP, we introduce a simple, randomized Decider/Chooser Protocol (DCP). DCP is a Las Vegas type randomized algorithm: the labels that are computed always satisfy the problem specification, but the algorithm is only probabilistically fast. It is also a fully dynamic algorithm [3], in that it makes use of previous solutions to solve the problem more quickly than by recomputing from scratch.

We have applied DCP to a variety of problems in large scale networks. For instance, DCP can be used for handoff in wireless networks, with mobile devices functioning as choosers and APs as deciders; this exemplifies a topology with frequently changing links between choosers and deciders. Additionally, by applying DCP to a portion of Alias [2], we drastically reduce the amount of forwarding state needed in network elements. We also apply a modified version of DCP in Alias, in which the chooser is distributed across multiple nodes. Doing so allows for hierarchical address aggregation, which in turn reduces the forwarding state maintained by network elements.

Assigning labels to nodes is not a new problem. Perhaps closest to our problem is [1], which considers the issues of assigning labels to nodes in an anonymous network of unknown size. With DCP, we leverage the symmetry inherent in our network topology and use randomization to design a more practical algorithm.

References

1. Fraigniaud, P., Pelc, A., Peleg, D., Perennes, S.: Assigning labels in unknown anonymous networks. In: Proceedings of the 19th Annual Symposium on Distributed Computing, pp. 101–111. ACM, New York (2000)
2. Walraed-Sullivan, M., Niranjan Mysore, R., Tewari, M., Zhang, Y., Marzullo, K., Vahdat, A.: Alias: Scalable, decentralized label assignment for data centers. To appear in SOCC 2011 (2011)
3. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarithmic time per operation. J. ACM 46, 502–516 (1999)

Brief Announcement: $\Delta\Omega$: Specifying an Eventual Leader Service for Dynamic Systems[*]

Mikel Larrea[1] and Michel Raynal[2]

[1] University of the Basque Country, UPV/EHU, Spain
mikel.larrea@ehu.es
[2] Institut Universitaire de France & IRISA, France
michel.raynal@irisa.fr

Abstract. The election of an eventual leader in an asynchronous system prone to process crashes is an important problem of fault-tolerant distributed computing. This problem is known as the implementation of the failure detector Ω. In this work we propose a specification of Ω suited to dynamic systems, i.e., systems in which processes can enter and leave.

1 System Model and Specification of $\Delta\Omega$

We assume that the system is made up of an arbitrary number of processes. We consider that i is the identity of the process denoted p_i. We also assume that there is a sequential time domain $\mathcal{T}$. This time, that is not known by the processes, is used only to describe the behavior of the system and specify the eventual leadership problem. Let $\tau \in \mathcal{T}$ and τ_0 be the time at which the system starts. $\Pi(\tau)$ denotes the set of processes that compose the system at time τ. Hence, $\Pi(\tau_0) \neq \emptyset$ is the initial set of processes. It is assumed that, at any time, there is at least one process in the system: $\forall \tau : \ \Pi(\tau) \neq \emptyset$.

Let us consider a process p_i that joins the system. This occurs at some time denoted $\tau(join_i)$. Similarly, if it ever crashes or leaves the system, its crash/departure time is denoted $\tau(exit_i)$. If p_i neither crashes nor leaves the system, $\tau(exit_i) = +\infty$.

Each process p_i has a local variable denoted $leader_i$ whose aim is to contain the identity of the leader. When $p_i \in \Pi(\tau)$, the notation $leader_i^\tau$ is used to denote the value of $leader_i$ at time τ.

Preliminary definitions

- Given a message m, $m.sender$ denotes the identity of its sender.
- $MSG\_inT(\tau)$ denotes the set of messages that are in transit at time τ.
- $LEFT(\tau)$ is the set of processes that have left the system or crashed by time τ.
- $SANE(\tau) \equiv \big[\forall m \in MSG\_inT(\tau) : \ m.sender \notin LEFT(\tau)\big]$.

[*] Mikel Larrea has been supported by the Spanish Research Council, grant TIN2010-17170, and the Basque Government, grants IT395-10 and S-PE10UN55. Michel Raynal has been supported by the SHAMAN French ANR project.

Formal statement of the EL_NI property (Eventual Leadership in Non-Increasing systems). If after some time τ, no process enters the system, a leader is eventually elected. This property classically defines Ω in non-dynamic systems [1].

$$\begin{aligned}&\big[\exists\ \tau : \forall\ \tau_1, \tau_2 \geq \tau : (\tau_1 \geq \tau_2) \Rightarrow (\Pi(\tau_1) \subseteq \Pi(\tau_2)\big]\\ &\quad\Rightarrow \big[\exists\ p_\ell \in \textstyle\bigcap_{\tau' \geq \tau} \Pi(\tau') : \exists\ \tau_1 \geq \tau : \forall\ \tau_2 \geq \tau_1 : \forall\ p_i \in \Pi(\tau_2) : leader_i^{\tau_2} = \ell\big].\end{aligned}$$

Formal statement of the EL_ND property (Eventual Leadership in Non-Decreasing systems). While EL_NI addresses the case where, after some time, the system only decreases, the property EL_ND is its counterpart for the case where after some time, the system only increases. If after some time τ, no process leaves the system or crashes, a leader has to be eventually elected (Item i), and the new processes eventually adopt it forever (Item ii).

$$\begin{aligned}&\big[\exists\tau :\ \forall\ \tau_1, \tau_2 \geq \tau : (\tau_1 \leq \tau_2) \Rightarrow (\Pi(\tau_1) \subseteq \Pi(\tau_2))\big]\\ &\quad\Rightarrow \big[\exists\tau_1 \geq \tau,\ \exists\ p_\ell \in \Pi(\tau_1) :\\ &\qquad\quad \text{(i)} \qquad \big[\forall\ \tau_2 \geq \tau_1 : \forall\ p_i \in \Pi(\tau) : leader_i^{\tau_2} = \ell\big]\\ &\qquad\quad \text{(ii)}\ \ \wedge\ \big[\forall\ \tau' \geq \tau : \big(p_i \in \Pi(\tau') \setminus \Pi(\tau)\big)\\ &\qquad\qquad\qquad\quad \Rightarrow \big(\exists\ \tau_0 \geq \tau(join_i) : \forall\ \tau'' \geq \tau_0 : leader_i^{\tau''} = \ell\big)\big]\\ &\qquad\big].\end{aligned}$$

Formal statement of the STAB property. The third property addresses the stability of the elected leader. It states that as soon as there is a time τ such that (1) all processes in $\Pi(\tau)$ agree on the same leader p_ℓ and (2) the system is sane (in the sense it cannot be polluted by old messages from processes that have left the system or have crashed) then any process p_i in $\Pi(\tau)$ continues to consider p_ℓ as its leader until p_i or p_ℓ exits the system by leaving or crashing (Item i). Moreover, if time permits (Item ii) all processes joining the system eventually consider p_ℓ as their leader. Let us observe that Item ii is not redundant with respect to the previous EL_NI and EL_ND properties, as it covers the case where p_ℓ remains forever in the system while processes are permanently entering and leaving the system. The interested reader will find more developments in [2].

$$\begin{aligned}&\big[\exists\ \tau,\ \exists p_\ell \in \Pi(\tau) : \big((\forall\ p_i \in \Pi(\tau) : leader_i^{\tau} = \ell) \wedge SANE(\tau)\big)\big]\\ &\quad\Rightarrow \big[\forall\ \tau' : \tau \leq \tau' \leq \tau(exit_\ell) :\\ &\qquad\qquad \text{(i)} \qquad \big[\forall\ p_i \in \Pi(\tau) \cap \Pi(\tau') : leader_i^{\tau'} = \ell\big]\\ &\qquad\qquad \text{(ii)}\ \ \wedge\ \big[\forall\ p_i \in \Pi(\tau') \setminus \Pi(\tau):\\ &\qquad\qquad\qquad \big[(\tau(exit_i) = \tau(exit_\ell) = +\infty)\\ &\qquad\qquad\qquad\quad \Rightarrow \big(\exists\ \tau'' \geq \tau(join_i) :\ \forall\tau''' \geq \tau'' :\ leader_i^{\tau'''} = \ell\big)\big]\\ &\qquad\qquad\qquad\big]\\ &\qquad\big].\end{aligned}$$

References

1. Chandra, T.D., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. Journal of the ACM 43(4), 685–722 (1996)
2. Larrea, M., Raynal, M.: Specifying and Implementing an Eventual Leader Service for Dynamic Systems. Technical Report 1962, IRISA (France) (2010)

Brief Announcement: The BG-Simulation for Byzantine Mobile Robots*

Taisuke Izumi[1], Zohir Bouzid[2], Sébastien Tixeuil[2,3], and Koichi Wada[1]

[1] Graduate School of Engineering, Nagoya Institute of Technology, Nagoya, 466-8555, Japan
t-izumi@nitech.ac.jp
[2] Université Pierre et Marie Curie - Paris 6, LIP6 CNRS 7606, France
[3] Institut Universitaire de France

Motivation. Robot networks [4] have recently become a challenging research area for distributed computing researchers. At the core of scientific studies lies the characterization of the minimum robots capabilities that are necessary to achieve a certain kind of tasks, such as the formation of geometric patterns, scattering, gathering, *etc.* The considered robots are often very weak: They are anonymous, oblivious, disoriented, and most importantly dumb. The last property means that robots cannot communicate explicitly by sending messages to one another. Instead, their communication is indirect (or spatial): a robot 'writes' a value to the network by moving toward a certain position, and a robot 'reads' the state of the network by observing the positions of other robots in terms of its *local coordinate system.* The problem we consider in this paper is the *gathering* of fault-prone robots. Given a set of oblivious robots with arbitrary initial locations and no agreement on a global coordinate system, the gathering problem requires that all correct robots reach and stabilize the same, but unknown beforehand, location. A number of solvability issues about the gathering problem are studied in previous work because of its fundamental importance in both theory and practice. One can easily find an analogy of the gathering problem to the *consensus* problem, and thus may think that its solvability issue are straightforwardly deduced from the known results about the consensus solvability (e.g., FLP impossibility). However, many differences lies between those two problems and the solvability of the gathering problem is still non-trivial. An important witness encouraging the difference is that the gathering problem can be solved in a certain kind of crash-prone asynchronous robot networks [1,3], while the consensus cannot be solved under the asynchrony and one crash fault.

Our Contribution. In this paper, we investigate the solvability of the gathering problem of n-robot networks subject to *Byzantine* faults. As we mentioned, there still exists a large gap between possibility and impossibility of the Byzantine gathering problem. As known results, Byzantine gathering is feasible only under very strong assumptions (fully-synchronous atomic-execution models or

* This work is supported in part by the DIGITEO project PACTOLE, the ANR projects SHAMAN and R-DISCOVER, KAKENHI no. 22700010 and 21500013, and Foundation for the Fusion of Science and Technology (FOST).

small number of robots) [1], and also the impossibility results are proved only for severe models (asynchrony, oblivious and uniform robots, and/or without agreement of coordinate systems) [1,3]. Filling this gap has remained an open question until now. In this paper, we respond negatively: Namely, we prove that Byzantine gathering is impossible even if we assume an ATOM models, n-bounded centralized scheduler, non-oblivious and non-uniform robots, and a common orientation of local coordinate systems, for only one Byzantine robot. Those assumptions are much stronger than that shown in previous work, inducing a much stronger impossibility result.

At the core of our impossibility result is a reduction to 1-Byzantine-resilient gathering in mobile robot systems from the distributed 1-crash-resilient consensus problem in asynchronous shared-memory systems. In more details, based on the distributed BG-simulation by Borowsky and Gafni [2], we newly construct a 1-crash-resilient consensus algorithm using any 1-Byzantine-resilient gathering algorithm on the system with several constraints. Thus, we can deduce impossibility results of Byzantine gathering for the model stated above. More interestingly, because of its versatility, we can easily extend our impossibility result for general pattern formation problems: We show that the impossibility also holds for a broad class of pattern formation problems including line and circle formation. To the best of our knowledge, this paper is the first study explicitly bridging algorithmic mobile robotics and conventional distributed computing theory for proving impossibility results.

It is remarkable that our reduction scheme equips a non-trivial feature which is not addressed in the prior work of the BG-simulation: It privides a “synchrony” simulation on the top of fully-asynchronous shared memory systems. The assumption of n-bounded scheduler restricts the relative speed of each robot (formally, n-bounded scheduler only allows the activation schedules where each robot is activated at most n times between any two consecutive activations of some robot). An interesting insight we can find from our result is that it is possible to trade the synchrony and Byzantine behavior of robot networks to the asynchrony and crash behavior of shared memory systems, which implies that the gap between synchronous robot networks and classical distributed computation models is as large as that between synchrony and asynchrony in classical models.

References

1. Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. SIAM Journal on Computing 36(1), 56–82 (2006)
2. Borowsky, E., Gafni, E., Lynch, N.A., Rajsbaum, S.: The BG distributed simulation algorithm. Distributed Computing 14(3) (2001)
3. Defago, X., Gradinariu, M., Messika, S., Parvedy, P.: Fault-tolerant and self-stabilizing mobile robots gathering. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 46–60. Springer, Heidelberg (2006)
4. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal on Computing 28(4), 1347–1363 (1999)

DISC 2011 Invited Lecture: Polygon Reconstruction with Little Information: An Example for the Power of Simple Micro-robots

Peter Widmayer

Department of Computer Science, ETH Zurich, Zurich, Switzerland

Abstract. For which settings will local observations of an unknown environment allow simple mobile agents (micro-robots) to draw global conclusions? This question has been studied for a long time when the environment is a graph, and the mobile agents walk on its vertices, under a variety of models. In this talk, however, we are interested in geometric environments.

More specifically, we discuss the problem of reconstructing an unknown simple polygon from a series of local observations. We aim to understand what types of sensing information acquired at vertices of the polygon carry enough information to allow polygon reconstruction by mobile agents that move from vertex to vertex. It turns out that ideas from distributed computing help to reconstruct the polygon topology even if the sensing information is purely non-geometric. We also briefly touch on a few related problems such as guarding the polygon and rendezvous. The reported work has been done over the years with Davide Bilo, Jeremie Chalopin, Shantanu Das, Yann Disser, Beat Gfeller, Matus Mihalak, Subhash Suri, and Elias Vicari.

Locality and Checkability in Wait-Free Computing[⋆]

Pierre Fraigniaud[1,⋆⋆], Sergio Rajsbaum[2,⋆⋆⋆], and Corentin Travers[3,†]

[1] CNRS and U. Paris Diderot, France
pierre.fraigniaud@liafa.jussieu.fr
[2] Instituto de Matemáticas, Universidad Nacional Autónoma de México, Mexico
rajsbaum@math.unam.mx
[3] LaBRI, U. of Bordeaux and CNRS, France
travers@labri.fr

Abstract. This paper studies notions of locality that are inherent to the specification of a distributed task and independent of the computing environment, in a shared memory wait-free system.

A locality property called *projection-closed* is identified, that completely characterizes tasks that are wait-free *checkable*. A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is checkable if there exists a wait-free distributed algorithm that, given $s \in \mathcal{I}$ and $t \in \mathcal{O}$, determines whether $t \in \Delta(s)$, i.e., if t is a valid output for s according to the specification of T. Moreover, determining whether a projection-closed task is wait-free solvable remains undecidable, and hence this is a rich class of tasks.

A stronger notion of locality considers tasks where the outputs look identically to the inputs at every vertex (input value of a process). A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is said to be *locality-preserving* if $\mathcal{O}$ is a covering complex of $\mathcal{I}$. This topological property yields obstacles for wait-free solvability different in nature from the classical agreement impossibility results. On the other hand, locality-preserving tasks are projection-closed and therefore always wait-free checkable. A classification of locality-preserving tasks in term of their relative computational power is provided. A correspondence between locality-preserving tasks and subgroups of the *edgepath* group of an input complex shows the existence of hierarchies of locality-preserving tasks, each one containing at the top the universal task (induced by the universal covering complex), and at the bottom the trivial identity task.

Keywords: distributed verification, local computing, wait-free, decision task.

⋆ Work supported in part by a CONACYT/CNRS grant.
⋆⋆ Additional supports from the ANR projects ALADDIN and PROSE, and from the INRIA project GANG.
⋆⋆⋆ Additional supports from UNAM-PAPIIT and PAPIME.
† Work done in part in U. Pierre et Marie Curie. Additional supports from ANR project SPREAD and INRIA project REGAL.

1 Introduction

A *task* is a distributed coordination problem in which each process starts with a private input value, communicates with the other processes by applying operations to shared objects, and eventually decides a private output value. It can be described by a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ is the set of input configurations, $\mathcal{O}$ is the set of output configurations, and Δ is the specification of the task mapping every input configuration to a set of possible output configurations. A *protocol* is a distributed program that solves a task.

Often it is useful to consider (input or output) configurations where the states of only a subset of the processes are specified. If s is an input configuration, $\pi(s)$ denotes the configuration obtained by projecting out the processes specified by π. Then, the map Δ of a task specifies with $\Delta(\pi(s))$ the valid output configurations for $\pi(s)$. For instance, let G be a network of processes, with one process per node, exchanging information along their incident edges. Assume one wants to color the nodes of G in such a way that two adjacent nodes are assigned different colors. The literature tackling this task (see, e.g., [6,29]) generally assumes that $d+1$ colors are available, where d denotes the maximum degree of G. The main motivation for this assumption is that every graph is $(d+1)$-colorable, whereas there are graphs that are not d-colorable, like, e.g., the complete graphs, or the cycles of odd length. A more careful look at the specification Δ of the $(d+1)$-coloring task enables to identify a stronger property: any partial $(d+1)$-coloring $\Delta(\pi(s))$ for a subgraph $\pi(s)$ induced by any set of processes specified by π, can be extended to a $(d+1)$-coloring $\Delta(s)$ for s. In other words, the $(d+1)$-coloring specification Δ satisfies the *monotony* condition

$$\Delta(\pi(s)) \subseteq \pi(\Delta(s)) \tag{1}$$

for every set of processes s, and every projection π. Instead, the specification of the d-coloring task does not satisfy this inclusion, even in networks that are d-colorable. For instance, if s denotes the four nodes of a 4-cycle, and $\pi(s)$ denotes two antipodal nodes in this cycle, then the output consisting in coloring the nodes of $\pi(s)$ with distinct colors cannot be extended to a valid output for the whole set s. It may thus be not coincidental that 3-coloring the n-node ring can be achieved in $O(\log^* n)$ rounds (see [10]) whereas 2-coloring rings (of even size) requires $\Omega(n)$ rounds (see [30]).

The monotony condition expresses a notion of *locality* satisfied by the task, that can be phrased as: *any output for a partial input is a partial output for the full input.* It is important to observe that this notion expresses a form of locality that is inherent to the specification of a task, and independent of the distributed computing model. For instance, at the other extremity of the wide spectrum of distributed computing models, previous work in wait-free computing, where asynchronous processes subject to crash failures communicate via a read/write shared memory, often assumes tasks satisfying the monotony condition. Typically, consensus satisfies it. (Note that the inclusion is strict for consensus, whereas equality holds for $(d+1)$-coloring). Monotony captures locality

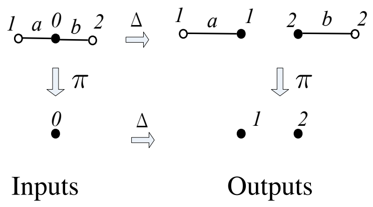


Fig. 1. An intersection-closed task, which is not monotone, and not wait-free solvable. The projection π depicted is for the black process. The black process always starts with 0. When both run, they must agree on the input of the white process, either 1 or 2. If the white process runs solo, it must decide its own input. If the black process runs solo, it can decide 1 or 2.

in a general sense, independent of the computing environment, by expressing the relationship between the various scales of computation. Indeed, monotony relates the specification for subsets of processes to the specification for larger sets. Thus, it relates the individual behavior of each process with the behavior of small group of processes, which in turn is related to the behavior of larger and larger groups, until one reaches the scale of the whole system.

A weaker form of locality results from putting the burden on the protocol to find a right output in $\Delta(\pi(s))$ for $\pi(s)$, that can be extended to an output $\Delta(s)$ for s. In other words, instead of imposing a task to satisfy Eq. 1, it might be sufficient to assume the *intersection-closeness* condition

$$\Delta(\pi(s)) \cap \pi(\Delta(s)) \neq \emptyset. \tag{2}$$

This weaker condition is an obvious requirement for a task to be wait-free solvable whereas monotony is not a necessary condition for wait-free solvability. However, putting this burden on the shoulders of the protocol may be too much. Indeed, for $s \neq s'$ with $\pi(s) = \pi(s')$, it may be the case that $\pi(\Delta(s)) \cap \pi(\Delta(s')) = \emptyset$ even if Eq. 2 is satisfied for all s, s', and π. See Fig. 1 for an example.

In this case, the processes in $\pi(s)$ running alone have no clue whether they have to output a solution extendable to an output for s or for s'. This is why tasks are usually assumed to satisfy the monotony condition instead of the weaker intersection-closeness condition.

Unfortunately, neither the intersection-closeness condition nor even the monotony condition provide sufficient constraints for solvability, or for efficient computation. For instance, in the network setting, monotony is not a guaranty for a task to be solved by having every node merely inspecting nodes at a constant distance (cf. the $\Omega(\log^* n)$ lower bound for $(d+1)$-coloring [30]). Neither it is, in the wait-free setting, a guaranty for a task to be solvable (cf. the FLP impossibility result for consensus [14]). This paper investigates two other notions of locality, namely the *projection-closeness* condition, obtained by simply reversing the monotony condition,

$$\pi(\Delta(s)) \subseteq \Delta(\pi(s)) \tag{3}$$

and the one resulting from combining monotony with projection-closeness. These latter two notions are shown to be rich concepts, enabling to capture important features of the computational nature of tasks, at least as far as wait-free computing is concerned.

Our Results. The objective of this paper is to investigate the ability of a shared memory system to solve tasks satisfying various forms of locality. Our investigation is performed in the wait-free setting where computation tolerates the halting failures or delays by $n-1$ out of n processes. In this context, it is convenient to view $\mathcal{I}$ and $\mathcal{O}$ as complexes (including configurations for *any* number of processes, between 1 and n), with the specification Δ mapping every simplex $s \in \mathcal{I}$ to a sub-complex $\Delta(s)$ of $\mathcal{O}$. See Figure 2 for a graphical representation of our results.

First, we show that the projection-closeness condition of Eq. 3 is closely related to the ability of *checking* a task. Informally, for checking a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$, every process $i \in [n]$ is given a pair (s_i, t_i), and the participating processes must check that the simplex t with decision values t_i, $i = 1, \dots, n$, is a valid output simplex according to Δ, for an input simplex s with values s_i. Deciding the latter is performed according to the (informal) specifications:

- if $t \in \Delta(s)$ then all participating processes must output “yes”,
- otherwise at least one participating process must output “no”.

Hence checking the task T corresponds to solving a *checking task*, where the input entry of the ith process is a pair $(i, (s_i, t_i))$, and where each process must return either “yes” or “no”. We prove that a task is wait-free checkable if and only if it is projection-closed (cf. Theorem 1). It is remarkable that the locality notion expressed by Eq. 3 captures precisely the ability to wait-free verify the results of a computation, even if these results have been obtained using more resources (e.g., oracles) or stronger models (e.g., t-resilience). Moreover, we show that the set of projection-closed tasks is large by proving that determining whether a projection-closed task is wait-free solvable remains undecidable (cf. Theorem 2). This latter result is obtained by proving that every task is equivalent to a wait-free checkable task (via implementations that preserve step-complexity).

Next, we turn our attention to tasks that are both projection-closed and monotone. As for monotony alone, the two conditions combined do not seem to provide sufficient structural constraints for relating them to wait-free computability. Nevertheless, we were able to identify a subclass of these tasks, that offers a stronger notion of locality expressible in the framework of algebraic topology. A task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is said to be *locality-preserving* if and only if $\mathcal{O}$ is a *covering complex* of $\mathcal{I}$, that is there exists a map $p : \mathcal{O} \to \mathcal{I}$ which agrees with Δ, i.e.,

$$\exists p : \mathcal{O} \to \mathcal{I} \mid \forall t \in \mathcal{O}, t \in \Delta(p(t)). \tag{4}$$

We show that, indeed, locality-preserving form a subclass of the monotone and projection-closed tasks (cf. Theorem 3). The notion of locality captured by locality-preserving tasks is made explicit topologically. Informally, locality-

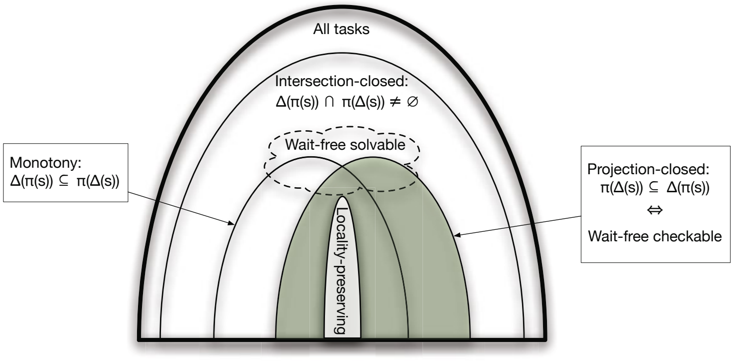


Fig. 2. The universe of tasks

preserving tasks are tasks where, from the perspective of a subset $\pi(s)$ of processes with given input values, the possible inputs to the other processes look identically in structure to their possible outputs (see Figure 3 for an example).

We show that locality-preserving tasks form a wide and rich class of tasks. We *classify* locality-preserving tasks in terms of their computational power. By identifying a correspondence between locality-preserving tasks and covering complexes (the classic algebraic topology notion used to study locality), we prove that locality-preserving task T implements locality-preserving task T' if and only if $H \subseteq H'$, where H, H' are groups associated to each of the tasks. This result demonstrates the existence of an infinite set of partial orders of locality preserving tasks. Each of these partial orders contains a hierarchy of locality-preserving tasks between the trivial identity task, and a universal task for this partial order. Some of these partial orders are finite, while others are infinite. As in [25], we use topology techniques both to prove impossibility results, and to show when one task can implement another.

Due to space limitation, proofs are presented in a companion technical report [17].

Related Work. The locality notions considered in this paper, are local in the sense that they can be checked individually for pairs (s,t), $s \in \mathcal{I}$, $t \in \mathcal{O}$. The main obstacles to wait-free solvability studied in the past, most notably for set agreement and *renaming* [2], are of a different nature. Indeed, any wait-free protocol is actually a mapping from a subdivision of the input complex to the output complex [26]. Hence, topological properties must be preserved by a wait-free protocol. Checking these properties is hard, and, in fact, determining whether a task is wait-free solvable is not decidable [20,24].

A different notion of locality has received lots of attention in the framework of network computing. Specifically, the so-called $\mathcal{LOCAL}$ and $\mathcal{CONGEST}$ models [35] have been designed to study communication locality issues. One prominent result in this framework is the $\Omega(\log^* n)$ lower bound [30] for the number of rounds required to 3-color the nodes of the n-node ring network. In several papers in this framework, the main focus is on whether randomization helps [34],

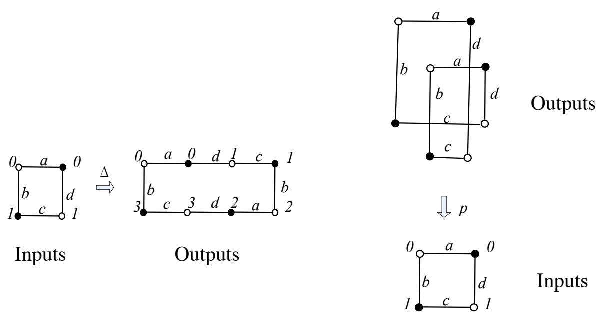


Fig. 3. Two-cover task. The task is for two processes, which start with binary inputs. On the left side of the figure, the specification of the task says that if both start with the same value, they must decide the same value: if they start with 0, decide either 0 or 2; if they start with 1, decide 1 or 3. If they start with different values, the valid outputs are defined in the figure, via edge labels b or d. The relation Δ defines also the possible outputs when only one process runs solo: e.g., when the white process starts with 0, it can decide 0 or 2, while if it starts with 1, it can decide 1 or 3. Notice that $\mathcal{O}$, a cycle of length 8, locally looks like $\mathcal{I}$, a cycle of length 4, in the sense that the 1-neighborhood of each vertex v in $\mathcal{I}$ is identical to the 1-neighborhood of a corresponding vertex in $\Delta(v)$. In the right side of the figure it is shown how $\mathcal{O}$ covers $\mathcal{I}$, by wrapping around it twice, where p identifies edges with the same label.

on the impact of non-determinism [16], and on the power of oracles providing nodes with information about their environment [15]. The impact on the design of efficient protocols of the absence of a priori knowledge on the global environment has been recently addressed in [28].

Starting with the pioneering work by Angluin [1], covering spaces have been used to derive impossibility results in *anonymous networks*, but only in the 1-dimensional case of graph coverings. Sufficient and sometimes necessary conditions on the communication graph and on the initial common knowledge for solving fundamental distributed problems such as leader election or termination detection are given in, e.g., [1,9,33], under several models of local computation. See [9] for an introduction to local computation in anonymous networks.

One can roughly classify the methods for ensuring the correctness of a program as either *verifying*, *testing*, or *checking*. Unlike verifying and testing, checking is performed at run time. A sequential checker [7] consists of a battery of tests (performed at run time) which compare the output of the program with either a predetermined value, or with a function of the outputs of the same program corresponding to different inputs. A similar idea is *spot-checking* [13] where the goal it to know if the output is reasonably correct, i.e., close in some problem-specific distance to the correct output. Related areas may be *learning*, where samples of outputs are used to infer the task it is being solved, as in [19] and *property testing* [21]. Blum et al. [8] introduced the notion of program *testers* and *correctors*, see also [22,31].

In the parallel and/or distributed computing context, the results are not so advanced as in the sequential setting. Parallel program checking has been studied in the PRAM model [37]. In the synchronous model, distributed self-testing and correcting protocols that tolerate crash failures are presented for the byzantine generals task in [18]. Self-testing/correcting is reminiscent of the notion of checking as a means of making a distributed algorithm *self-stabilizing*, as explored in [4,5] in the synchronous setting. In the framework of network computing, distributed verification has been addressed only recently (see, e.g., [11]), though previous research on proof labeling schemes [27] already gave some insights on the ability of checking global predicates locally (see also [23]).

2 Model

We consider a standard read/write shared memory wait-free model. The system consists of n asynchronous processes, denoted by the integers in $[n] = \{1, \ldots, n\}$. The processes can fail by crashing (a crashed process never recovers). Any number of processes can crash, at any time. We recall here the main features of this model, and refer to [3,26,32] for a more detailed and accurate description.

The processes that take an infinite number of steps in a run are correct, the others crashed. If a process crashes initially, it does not take any step, and we say it does not participate in the run. A process participates in a run if it takes at least one step. At the core of the model is the following assumption. We enforce protocols to be *wait-free*, that is to avoid all instructions that would cause a process to wait for the result of the action of another process. In particular, in a wait-free protocol, a process i cannot check whether another process has crashed, or is just very slow.

When solving a task, in each run of a protocol, processes start with private input values. A process $i \in [n]$ is not aware of the inputs of other processes. The initial states of the processes differ only in their input values. Each process i has to eventually decide irrevocably on a value. Consider a run r where only a subset of k processes participate, $1 \leq k \leq n$. These processes have distinct identities $\{\mathrm{id}_1, \ldots, \mathrm{id}_k\}$, where for every $i \in [k]$, $\mathrm{id}_i \in [n]$. A set $s = \{(\mathrm{id}_1, x_1), \ldots, (\mathrm{id}_k, x_k)\}$, is used to denote the input values or decision values in the run, where x_i denotes the value of the process with identity id_i (either an input value, or a decision value). We denote by $\mathrm{ID}(s)$ the set of identities of the processes in s, i.e., $\mathrm{ID}(s) = \{\mathrm{id}_1, \ldots, \mathrm{id}_k\}$. The input values of processes not in $\mathrm{ID}(s)$ are irrelevant, as they do not participate in the run, so they are not included in s.

Let s' be a subset of $s = \{(1, x_1), \ldots, (n, x_n)\}$, $\mathrm{ID}(s) = [n]$. We say that $\pi(s) = s'$, for a *projection* π, that eliminates processes in $\mathrm{ID}(s) \setminus \mathrm{ID}(s')$. Because any number of processes can crash, all such subsets s' of s are of interest, to consider runs where only processes in $\mathrm{ID}(s')$ may participate. That is, the set of possible input sets forms a *complex* because its sets are closed under containment. Similarly, the set of possible output sets also form a complex. Following the topology notation, the sets of a complex are called *simplexes*.

More formally, a *complex* K is a set of vertices $V(K)$, and a family of finite, nonempty subsets of $V(K)$, called *simplexes*, satisfying: (i) if $v \in V(K)$ then $\{v\}$ is a simplex, and (ii) if s is a simplex, so is every nonempty subset of s. The *dimension* of a simplex s is $|s|-1$, the dimension of K is the largest dimension of its simplexes, and K is *pure* of dimension k if every simplex belongs to a k-dimensional simplex. A 1-dimensional complex K is thus simply a graph[1]. In distributed computing, one refers to *colored* simplexes (and complexes), since each vertex v of a simplex is labeled with a distinct process identity $i \in [n]$.

We denote by $\mathcal{I}$ the input complex, and by $\mathcal{O}$ the output complex. An *input-output pair* is a pair (s,t) made of a simplex $s \in \mathcal{I}$ and a simplex $t \in \mathcal{O}$, with $\mathrm{ID}(t) \subseteq \mathrm{ID}(s)$. The strict inclusion $\mathrm{ID}(t) \subset \mathrm{ID}(s)$ takes into account the case when the processes in $\mathrm{ID}(s) \setminus \mathrm{ID}(t)$ fail, and do not decide, which, in the wait-free model, should not prevent the participating non-failing processes to decide. A *task* T is described by a triple $(\mathcal{I},\mathcal{O},\Delta)$ where $\mathcal{I}$ and $\mathcal{O}$ are pure $(n-1)$-dimensional complexes, and Δ is a map from $\mathcal{I}$ to the set of non-empty sub-complexes of $\mathcal{O}$, satisfying (s,t) is an input-output pair for every $t \in \Delta(s)$. Intuitively, Δ specifies, for every simplex $s \in \mathcal{I}$, the valid outputs for the processes in $\mathrm{ID}(s)$ that may participate in the computation. We assume that Δ is (sequentially) computable.

A protocol $\mathcal{A}$ *solves* task $T = (\mathcal{I},\mathcal{O},\Delta)$ if, for every simplex $s \in \mathcal{I}$, and every run r of $\mathcal{A}$ on s, every correct process decides, and the simplex t corresponding to these decisions belongs to $\Delta(s)$.

3 Projection-Closeness and Wait-Free Checkability

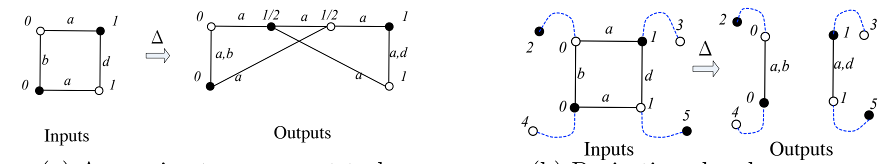
This section addresses the first of the two notions of locality tackled in this paper. The locality considered here is obtained by simply reversing the direction of the inclusion in the monotony condition of Eq. 1.

Definition 1. *A task $T = (\mathcal{I},\mathcal{O},\Delta)$ is* projection-closed *if and only if for every $s \in \mathcal{I}$ and every projection π, we have $\pi(\Delta(s)) \subseteq \Delta(\pi(s))$.*

Projection-closeness is as poorly related to wait-free solvability as is the intersection-closeness notion of Eq. 2. Consider for example, approximate agreement [12] for two processes, illustrated in Fig. 4a. While it is wait-free solvable, it is not projection-closed. Intuitively, because the *validity* requirement in dimension 0 "if a process runs solo, has to decide its own value" conflicts with the *agreement* requirement in dimension 1 "when processes start with different values, they can decide any values, as long as they are at most 1/2 apart from each other." More precisely, take the input simplex $s = \{(1,0),(2,1)\}$ (top edge

[1] It is the graph whose nodes are the vertices of K, and whose edges and nodes are the simplexes of K. More generally, the concept of complexes is, in some sense, a natural extension of the concept of graphs, to higher dimensions. They can also be viewed as forming a subclass of *hypergraphs*, in which every non-empty subset of an hyperedge must be an hyperedge.

labeled a in the figure), and apply the projection π that eliminates process 2, to obtain $\pi(\Delta(s))$ which consists of all vertices for process 1 in the output complex, while $\Delta(\pi(s))$ consists of only vertex $t = \{(1,0)\}$. For the same reason, consensus is not projection-closed; in contrast, consensus is not wait-free solvable.



(a) Approximate agreement task (b) Projection-closed consensus

Fig. 4. Two tasks for 2 processes

Instead, projection-closeness is well related to *checking* the result of a computation supposedly solving a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$. Intuitively, processes get as inputs the vertices of s, t, $s \in \mathcal{I}$, $t \in \mathcal{O}$, and want to decide if indeed the computation of t for s was correct, namely, if $t \in \Delta(s)$. That is, each process i gets as input a pair (x_i, y_i), such that the x_i values define s, while the y_i values define t. Then, checking T corresponds to solving a *checking task* T_c:

- If $t \in \Delta(s)$ then all processes that decide must output 1, interpreted as "yes".
- If $t \notin \Delta(s)$ then whenever *all* participating processes decide, one of them must output 0, interpreted as "no". When not all participating processes decide, then some may decide 1 and others 0.

Formally, $T_c = (\mathcal{I} \times \mathcal{O}, S^n, \Delta_c)$. The input complex $\mathcal{I} \times \mathcal{O}$ consists of all simplexes $s \times t$, $s \in \mathcal{I}$, $t \in \mathcal{O}$, $\text{ID}(s) = \text{ID}(t)$, where, for every $i \in [n]$,

$$(i, (x_i, y_i)) \in s \times t \iff \begin{cases} (i, x_i) \in s \\ (i, y_i) \in t \end{cases}$$

The output complex S^n consists of all simplexes where processes decide values in $\{0, 1\}$. Now, to define Δ_c, let S^J be the sub-complex of S^n induced by the processes in J, for $J \subseteq [n]$. In other words, $S^J = \pi(S^n)$ where π is the projection from $[n]$ to J. Also, for $J \subseteq [n]$, let $Y[J] = \{(i, 1), i \in J\}$ be the simplex corresponding to all processes in J outputting "yes", and let $\overline{Y}[J]$ be the complex induced by $Y[J]$. We have now all the ingredients to define Δ_c. For any $s \times t \in \mathcal{I} \times \mathcal{O}$ with $\text{ID}(s) = \text{ID}(t) = J$,

$$\Delta_c\Big(s \times t\Big) = \begin{cases} \overline{Y}[J] & \text{if } t \in \Delta(s) \\ S^J \setminus Y[J] & \text{otherwise.} \end{cases}$$

We now define wait-free checkability as follows:

Definition 2. *A task T is wait-free checkable if and only if its corresponding task T_c is wait-free solvable.*

The following theorem states the exact correspondence between projection-closeness and wait-free checkability.

Theorem 1. *A task T is wait-free checkable if and only if it is projection-closed.*

We now prove that the set of wait-free checkable tasks, or, equivalently, the set of projection-closed tasks, forms a large class of tasks.

Theorem 2. *Determining whether a wait-free checkable task is wait-free solvable is undecidable.*

To establish the theorem, we show that every task is essentially equivalent to a wait-free checkable task, under a very strong notion of equivalence. Following [25], a task T' *implements* task T (both defined on n processes) if one can construct a wait-free protocol for T by calling one instance of a black box for T' followed by any number of operations on read-write registers[2]. We write $T \leq T'$ to emphasize that T' is at least as powerful as T. In Section 4 we use this implementation, while for the theorem here, it suffices to use its particular case where no operations on read-write registers are used; namely, if there exists a wait-free protocol $\mathcal{A}$ that solves T, in which processes can call one instance of a black box that solves T', and do not execute any read-write operations:

Definition 3. *We say $T \leq T'$ if task T' implements task T, and $T \preceq T'$ if there is an implementation without executing any read-write operations.*

Based on Definition 3, one can define the following equivalence relation $\sim$ between tasks

$$T \sim T' \iff (T \preceq T' \text{ and } T' \preceq T)\ .$$

This equivalence notion and the fact that determining whether a task is wait-free solvable is undecidable [20,24] are the key ingredients in the proof of Theorem 2. Consider as an example consensus, which as mentioned above, is not wait-free checkable. However, consensus is equivalent to the *checkable consensus* task, depicted in Figure 4b for two processes. This task is obtained from consensus, by adding new input and output values $2, 3, 4, 5$, and then allowing for processes running solo on inputs $0, 1$, to decide any value (but when a process starts with $2, 3, 4$ or 5 it decides its own input always).

4 Locality-Preserving Tasks

In this section, we turn our attention to locality-preserving tasks, which are both monotone and projection-closed (see Fig. 2) and preserve locality in a strong, topological sense. Before defining the class, we need to recall some simple topology facts. Missing proof can be found in [17].

[2] A more general notion of implementation allows read-write operations before calling the black box, and calling the black box more than once.

Preliminaries. For two complexes K and K', a *map* $f : K \to K'$ is a function $f : V(K) \to V(K')$ such that whenever $s = \{v_0, \dots, v_q\}$ is a simplex in K, then $f(s) = \{f(v_0), \dots, f(v_q)\}$ is a simplex in K'. The map is *color-preserving* if it preserves ids, that is, for each $v \in V(K)$, $\mathrm{ID}(f(v)) = \mathrm{id}(v)$. Hence, if the map is color preserving, and $f(s) = s'$, then $dim(s) = dim(s')$. An *edge* e in a complex K is an ordered pair of (not necessarily) distinct vertices $e = (u, v)$, where $\{u, v\}$ is a simplex in K. The *origin* and *end* of $e = (u, v)$ are respectively denoted $orig(e) = u$ and $end(e) = v$. A *path* α in K is a finite sequence of edges $\alpha = e_1 \bullet e_2 \bullet \cdots \bullet e_k$, where $end(e_i) = orig(e_{i+1})$. The path α is *closed at* u if $orig(\alpha) = orig(e_1) = u = end(e_k) = end(\alpha)$. A complex K is *connected* if for every pair of vertices u, v in K, there is a path from u to v. A covering complex [36] is the discrete analogue of a covering space. The notion of covering complex formalizes the idea of one complex looking identically to another locally. Its definition is recalled below (see [36]):

Definition 4. *A pair* $(\tilde{K}, p)$ *is a* covering complex *of a complex* K *if and only if the following three properties are satisfied:*

1. $p : \tilde{K} \to K$ *is a map;*
2. $\tilde{K}$ *is connected;*
3. *for every simplex* s *in* K, $p^{-1}(s)$ *is a union of pairwise disjoint simplexes,* $p^{-1}(s) = \cup \tilde{s}_i$, *with* $p|\tilde{s}_i : \tilde{s}_i \to s$ *a one-one correspondence for each* i.

The simplexes $\tilde{s}_i$ are called the *sheets* over s. We often refer to p as a *covering map*. Next observations easily follow from the definition of covering complexes: $K = im(p)$, and hence, K is connected. If s is a q-simplex in K, each sheet $\tilde{s}_i$ over s is also a q-simplex. For each vertex v in K, the complex $star(v)$ consists of all simplexes that contain v. One can check that K and $\tilde{K}$ are *locally isomorphic* in the sense that if $\tilde{v}$ is such that $p(\tilde{v}) = v$, then $star(v)$ is isomorphic to $star(\tilde{v})$.

Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ be a task. In the following, we assume that the output complex is connected (otherwise, our analysis can be done on each connected component), and that $\mathcal{O}$ does not contain irrelevant simplexes. That is, for each $t \in \mathcal{O}$, there exists $s \in \mathcal{I}$ such that $t \in \Delta(s)$. Let $p : \mathcal{O} \to \mathcal{I}$ a covering map. We say that p *agrees with* Δ if $p(\tilde{s}) = s \iff \mathrm{ID}(s) = \mathrm{ID}(\tilde{s}) \wedge \tilde{s} \in \Delta(s)$, for every $s \in \mathcal{I}, \tilde{s} \in \mathcal{O}$. In particular, p is color-preserving.

Definition 5. *A task* $T = (\mathcal{I}, \mathcal{O}, \Delta)$ *is* locality-preserving *if and only if there exists a covering complex* $(\mathcal{O}, p)$ *of* $\mathcal{I}$, *with a map* p *that agrees with* Δ.

Example of locality-preserving tasks is depicted in Figure 3, where the color of a vertex (black or white) represents its identity. Instead of adding input and output values to vertexes, labels are added to edges, which is sufficient to specify Δ, as the tasks are both monotone and projection-closed. For example, in Figure 3, the edge labeled a on the left side represents an input simplex. This input simplex is mapped by Δ to a set of two output simplexes, namely the two edges labeled a on the right side of the picture. In these tasks, the outputs look like the inputs, locally. For example, in Figure 3, the top left corner formed by the white vertex (with the two edges labeled a and b) is mapped by Δ to the two opposite corners that look the same locally.

Characterization. Recall that task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ is said to be monotone when $\Delta(\pi(s)) \subseteq \pi(\Delta(s))$ for every $s \in \mathcal{I}$, and any projection π. In other words, for any $t' \in \Delta(\pi(s))$, there exists $t \in \Delta(s)$ such that $\pi(t) = t'$. We say that T is *strongly monotone* if, for every $s \in \mathcal{I}$, any projection π, and any $t' \in \Delta(\pi(s))$, there exists a *unique* $t \in \Delta(s)$ such that $\pi(t) = t'$. So, intuitively, a task that is strongly monotone permits to extend any partial output in a unique manner. In order to characterize locality-preserving, we say that a task T is *one-to-one* if and only if, for every pair $(s, s'), s \neq s'$ of 0-dimensional simplexes in $\mathcal{I}$, we have $\Delta(s) \cap \Delta(s') = \emptyset$.

Theorem 3. *A task T is locality-preserving if and only if T is projection-closed, one-to-one, and strongly monotone.*

As every projection-closed task is wait-free checkable (Theorem 1), we get the following.

Corollary 1. *Every locality-preserving task is wait-free checkable.*

As demonstrated by the next corollary, few locality-preserving tasks are wait-free solvable. For an input complex $\mathcal{I}$, the *identity task* $T_{Id,\mathcal{I}} = (\mathcal{I}, \mathcal{I}, \Delta)$ over $\mathcal{I}$ simply requires that each process decides its input in every execution: $\forall s \in \mathcal{I}, \Delta(s) = \{s\}$. More generally, we say that a task is an identity task by having each process output a function of its input, without any shared memory operations. Identity tasks with input $\mathcal{I}$ are the tasks equivalent to $T_{Id,\mathcal{I}}$ for the $\sim$ relation.

Corollary 2. *The identity tasks are the only locality-preserving task that are wait-free solvable.*

Hierarchies of Locality-Preserving Tasks. In this section we classify the locality-preserving tasks in term of their relative computing power, that is in their capacity of mutual implementation (Definition 3). For the remaining of this section, we fix an arbitrary input complex $\mathcal{I}$ and study the relative power of locality-preserving tasks with input $\mathcal{I}$. We establish that each such locality-preserving task induces subgroups of a group defined from the closed paths in $\mathcal{I}$. Moreover, the relative power of locality-preserving tasks with input $\mathcal{I}$ directly depends on the subgroups they induce.

Edgepath groups and locality-preserving tasks. Our exposition follows Rotman [36]. Let K a complex and $v_* \in V(K)$. The *edgepath group* of K with basepoint v_* is $G(K, v_*) = \{[\alpha] \ : \ \alpha \text{ is a closed path at } v_*\}$. $[\alpha]$ consists of the equivalence class of closed paths α' that can be deformed to α along 2-simplexes. More precisely, the paths α' and α are equivalent if one can be obtained from the other by applying the following rule a finite number of times: replace $(u, v) \bullet (v, w)$ by (v, w) or (v, w) by $(u, v) \bullet (v, w)$ whenever $\{u, v, w\}$ is a simplex of K. The group operation is path concatenation, which is compatible with path equivalence. Given $\alpha = e_1 \bullet \ldots \bullet e_n$, the inverse of $[\alpha]$ is $[\alpha^{-1}]$ where $\alpha^{-1} = e_n^{-1} \bullet \ldots \bullet e_1^{-1}$ and each e_i^{-1} is the edge obtained by reversing the end and origin of e_i. The identity element is $[(v_*, v_*)]$. If K is connected, changing the basepoint results in an isomorphic group ([36], Theorem 1.4).

Covering complexes of K induce subgroups of the edgepath group of K. Conversely, every subgroup of the edgepath group is induced by a covering complex. Formally, the notation of a covering complex is extended to *pointed complexes.* $p : (\tilde{K}, \tilde{v}_*) \to (K, v_*)$ is a covering complex when $(\tilde{K}, p)$ is a covering complex of K, and $p(\tilde{v}_*) = v_*$. Each covering complex $p : (\tilde{K}, \tilde{v}_*) \to (K, v_*)$ determines a subgroup of $G(K, v_*)$, namely, $p_\#(G(\tilde{K}, \tilde{v}_*))$, where $p_\#$ is the homomorphism induced by p, which is one-to-one ([36], Theorem 2.3). Let $H = p_\#(G(\tilde{K}, \tilde{v}_*))$. Keeping p unchanged but choosing a different basepoint $\tilde{v}'_* \in \tilde{K}$ such that $p(\tilde{v}'_*) = p(\tilde{v}_*) = v_*$ may induce a different subgroup $H' = p_\#(G(\tilde{K}, \tilde{v}'_*))$. H and H' are however *conjugate* subgroups of $G(K, v_*)$. Conversely, if H' is conjugate to H then $H' = p_\#(G(\tilde{K}, \tilde{u}))$ for some $\tilde{u} \in V(\tilde{K})$ ([36], Theorem 2.4). Finally, let $p : (\tilde{K}, \tilde{v}_*) \to (K, v_*)$ and $q : (\tilde{J}, \tilde{w}_*) \to (K, v_*)$ two covering complexes of (K, v_*) and $H(= p_\#(G(\tilde{K}, \tilde{v}_*))), H'(= q_\#(G(\tilde{J}, \tilde{w}_*)))$ the induced subgroups. If $H' \subseteq H$, there exists a unique map $r : (\tilde{J}, \tilde{w}_*) \to (\tilde{K}, \tilde{v}_*)$ such that $pr = q$. Moreover, $r : (\tilde{J}, \tilde{w}_*) \to (\tilde{K}, \tilde{v}_*)$ is a covering complex ([36], Theorem 3.3). In particular, when $H = H'$, r is an isomorphism.

On the other hand, for every subgroup H of $G(K, v_*)$ there exists a connected complex K_H and a map p such that $p : (K_H, \tilde{v}_*) \to (K, v_*)$ is a covering complex for some $\tilde{v}_* \in V(K_H)$ and $p_\#(K_H, \tilde{v}_*) = H$ ([36], Theorem 2.8). In particular, the trivial group $\{[(v_*, v_*)]\}$ that consists in the identity element is a subgroup of any subgroup, and the corresponding covering complex $p : (K_u, \tilde{v}_*) \to (K, v_*)$ is called the *universal covering* of K, because it covers any other covering of K. The universal covering complex is *simply connected* since $p_\#$ is one-to-one and $p_\#(G(K_u, \tilde{v}_*)) = \{1\}$.

By definition, each locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ determines a covering complex $p : \mathcal{O} \to \mathcal{I}$ and conversely, a covering complex $p : \mathcal{O} \to \mathcal{I}$ defines a locality-preserving task with input $\mathcal{I}$. It thus follows from the discussion above that every locality-preserving task with input $\mathcal{I}$ induces subgroups of the edgepath group of $\mathcal{I}$, and reciprocally each subgroup induces a locality-preserving task. This is captured by the next lemma.

Lemma 1. *Let $\mathcal{I}$ be a connected input complex and $v_* \in V(\mathcal{I})$. (1) Every subgroup $H \subseteq G(\mathcal{I}, v_*)$ induces a locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ whose associated covering map satisfies $p_\#(G(\mathcal{O}, \tilde{v}_*)) = H$ for some $\tilde{v}_* \in V(\mathcal{O})$. (2) Every locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ induces a conjugacy class of subgroups of $G(\mathcal{I}, v_*)$; each subgroup H in the class satisfies $H = p_\#(G(\mathcal{O}, \tilde{v}_*))$ for some $\tilde{v}_* \in V(\mathcal{O})$, where p is the covering map associated with T.*

Then, Theorem 3.3 in [36] discussed earlier implies the following result.

Lemma 2. *Let $\mathcal{I}$ be an input complex and $v_* \in V(\mathcal{I})$. Let $T = (\mathcal{I}, \mathcal{O}, \Delta)$ and $T' = (\mathcal{I}, \mathcal{O}', \Delta')$ two locality-preserving tasks with input $\mathcal{I}$ and $p : \mathcal{O} \to \mathcal{I}$, $p' : \mathcal{O}' \to \mathcal{I}$ their respective covering map. If $p'_\# G(\mathcal{O}', \tilde{v}'_*) \subseteq p_\#(G(\mathcal{O}, \tilde{v}_*))$ for some vertexes $\tilde{v}'_* \in V(\mathcal{O}'), \tilde{v}_* \in V(\mathcal{O})$, there exists a covering complex $p'' : (\mathcal{O}', \tilde{v}'_*) \to (\mathcal{O}, v_*)$ satisfying $p' = pp''$.*

Group-based hierarchies of locality-preserving tasks. We consider locality-preserving tasks that can be defined over a given input complex $\mathcal{I}$. The following result explicits the existence of a hierarchy of locality-preserving tasks, using the implementation relation "$\leq$" of Definition 3. By Lemma 1(2), every locality-preserving task induces a conjugacy class.

Theorem 4. *Let $\mathcal{I}$ be a connected complex. Let $T_{\mathcal{K}} = (\mathcal{I}, \mathcal{K}, \Delta_{\mathcal{K}})$ and $T_{\mathcal{L}} = (\mathcal{I}, \mathcal{L}, \Delta_{\mathcal{L}})$ be two locality-preserving tasks with input complex $\mathcal{I}$. Let $v_* \in V(\mathcal{I})$ and $C_{\mathcal{K}}$ and $C_{\mathcal{L}}$ the conjugacy classes of subgroups of $G(\mathcal{I}, v_*)$ induced by $T_{\mathcal{K}}$ and $T_{\mathcal{L}}$ respectively. $T_{\mathcal{L}} \leq T_{\mathcal{K}}$ if and only if $\exists H_{\mathcal{K}} \in C_{\mathcal{K}}, H_{\mathcal{L}} \in C_{\mathcal{L}}$ such that $H_{\mathcal{L}} \supseteq H_{\mathcal{K}}$.*

We say that a task is *universal* for some set of tasks if it implements any task in that set (in the sense of "$\leq$" in Definition 3). Theorem 4 implies that the set of locality-preserving tasks with input complex $\mathcal{I}$ has a universal task, which is the task defined by the universal covering of $\mathcal{I}$. More generally, it shows that every locality-preserving task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ lies in between the trivial identity task $T_{Id,\mathcal{I}}$ and the task defined by the universal covering complex of $\mathcal{I}$.

References

1. Angluin, D.: Local and Global Properties in Networks of Processors (Extended Abstract). In: 12th ACM Symp. on Theory of Computing (STOC), pp. 82–93 (1980)
2. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuck, R.: Renaming in Asynchronous Environment. Journal of the ACM 37(3), 524–548 (1990)
3. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley, Chichester (2004)
4. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by Local Checking and Correction. In: 32nd IEEE Symp. on Foundations of Computer Science (FOCS), pp. 268–277 (1991)
5. Awerbuch, B., Varghese, G.: Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols. In: 32nd IEEE Symp. on Foundations of Computer Science (FOCS), pp. 258–267 (1991)
6. Barenboim, L., Elkin, M.: Deterministic distributed vertex coloring in polylogarithmic time. In: 29th ACM Symp. on Principles of Distributed Computing (PODC), pp. 410–419 (2010)
7. Blum, M., Kannan, S.: Designing Programs that Check Their Work. J. ACM 42(1), 269–291 (1995)
8. Blum, M., Luby, M., Rubinfeld, R.: Self-Testing/Correcting with Applications to Numerical Problems. J. Comput. Syst. Sci. 47(3), 549–595 (1993)
9. Chalopin, J., Métivier, Y.: On the Power of Synchronization Between two Adjacent Processes. Distributed Computing 23(3), 177–196 (2010)
10. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control 70(1), 32–53 (1986)
11. Das Sarma, A., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed Verification and Hardness of Distributed Approximation. In: 43rd ACM Symp. on Theory of Computing, STOC (2011)
12. Dolev, D., Lynch, N., Pinter, S., Stark, E., Weihl, W.: Reaching Approximate Agreement in the Presence of Faults. J. ACM 33(3), 499–516 (1986)
13. Ergün, F., Kannan, S., Kumar, R., Rubinfeld, R., Viswanathan, M.: Spot-Checkers. J. Comput. Syst. Sci. 60(3), 717–751 (2000)

14. Fischer, M., Lynch, N., Paterson, M.: Impossibility of Distributed Consensus with One Faulty Process. J. ACM 32(2), 374–382 (1985)
15. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Communication algorithms with advice. J. Comput. Syst. Sci. 76(3-4), 222–232 (2010)
16. Fraigniaud, P., Korman, A., Peleg, D.: Local Distributed Decision. arXiv:1011.2152
17. Fraigniaud, P., Rajsbaum, S., Travers, C.: Locality and Checkability in Wait-free Computing. technical report, `http://hal.archives-ouvertes.fr/hal-00605244/en/`
18. Franklin, M., Garay, J.A., Yung, M.: Self-Testing/Correcting Protocols. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 269–284. Springer, Heidelberg (1999)
19. Freund, Y., Kearns, M., Ron, D., Rubinfeld, R., Schapire, R., Sellie, L.: Efficient Learning of Typical Finite Automata from Random Walks. Inf. Comput. 138(1), 23–48 (1997)
20. Gafni, E., Koutsoupias, E.: Three-Processor Tasks Are Undecidable. SIAM J. Comput. 28(3), 970–983 (1999)
21. Goldreich, O., Goldwasser, S., Shari, Ron, D.: Property Testing and its Connection to Learning and Approximation. J. ACM 45(4), 653–750 (1998)
22. Goldwasser, S., Gutfreund, D., Healy, A., Kaufman, T., Rothblum, G.: A (De)constructive Approach to Program Checking. In: 40th ACM Symp. on Theory of Computing (STOC), pp. 143–152 (2008)
23. Goos, M., Suomela, J.: Locally checkable proofs. In: 30th ACM Symp. on Principles of Distributed Computing, PODC (2011)
24. Herlihy, M., Rajsbaum, S.: The Decidability of Distributed Decision Tasks. In: 29th ACM Symp. on the Theory of Computing (STOC), pp. 589–598 (1997)
25. Herlihy, M., Rajsbaum, S.: A Classification of Wait-Free Loop Agreement Tasks. Theor. Comput. Sci. 291(1), 55–77 (2003)
26. Herlihy, M., Shavit, N.: The Topological Structure of Asynchronous Computability. J. ACM 46(6), 858–923 (1999)
27. Korman, A., Kutten, S., Peleg, D.: Proof Labeling Schemes. Distributed Computing 22, 215–233 (2010)
28. Korman, A., Sereni, J.-S., Viennot, L.: Toward more Localized Local Algorithms: Removing Assumptions concerning Global Knowledge. In: 30th ACM Symp. on Principles of Distributed Computing, PODC (2011)
29. Kuhn, F., Wattenhofer, R.: On the complexity of distributed graph coloring. In: 25th ACM Symp. on Principles of Distributed Computing (PODC), pp. 7–15 (2006)
30. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. 21(1), 193–201 (1992)
31. Lipton, R.: New Directions in Testing. In: DIMACS Workshop on Distributed computing and Cryptography, vol. 2, pp. 191–202 (1991)
32. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
33. Mazurkiewicz, A.: Distributed Enumeration. Inf. Process. Lett. 61, 233–239 (1997)
34. Naor, M., Stockmeyer, L.: What can be Computed Locally? SIAM J. Comput. 24(6), 1259–1277 (1995)
35. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM, Philadelphia (2000)
36. Rotman, J.: Covering Complexes with Applications to Algebra. Rocky Mountain J. of Mathematics 3(4), 641–674 (1973)
37. Rubinfeld, R.: Designing Checkers for Programs that Run in Parallel. Algorithmica 15(4), 287–301 (1996)

The Contest between Simplicity and Efficiency in Asynchronous Byzantine Agreement

Allison Lewko[⋆]

The University of Texas at Austin
alewko@cs.utexas.edu

Abstract. In the wake of the decisive impossibility result of Fischer, Lynch, and Paterson for deterministic consensus protocols in the aynchronous model with just one failure, Ben-Or and Bracha demonstrated that the problem could be solved with randomness, even for Byzantine failures. Both protocols are natural and intuitive to verify, and Bracha's achieves optimal resilience. However, the expected running time of these protocols is exponential in general. Recently, Kapron, Kempe, King, Saia, and Sanwalani presented the first efficient Byzantine agreement algorithm in the asynchronous, full information model, running in polylogarithmic time. Their algorithm is Monte Carlo and drastically departs from the simple structure of Ben-Or and Bracha's Las Vegas algorithms.

In this paper, we begin an investigation of the question: to what extent is this departure necessary? Might there be a much simpler and intuitive Las Vegas protocol that runs in expected polynomial time? We will show that the exponential running time of Ben-Or and Bracha's algorithms is no mere accident of their specific details, but rather an unavoidable consequence of their general symmetry and round structure. We view our result as a step toward identifying the level of complexity required for a polynomial-time algorithm in this setting, and also as a guide in the search for new efficient algorithms.

1 Introduction

Byzantine agreement is a fundamental problem in distributed computing, first posed by Pease, Shostak, and Lamport [18]. It requires n processors to agree on a bit value despite the presence of failures. We assume that at the outset of the protocol, an adversary has corrupted some t of the n processors and may cause these processors to deviate arbitrarily from the prescribed protocol in a coordinated malicious effort to prevent agreement. Each processor is given a bit as input, and all good (i.e. uncorrupted) processors must reach agreement on a bit which is equal to at least one of their input bits. To fully define the problem, we must specify the model for communication between processors, the computational power of the adversary, and also the information available to the adversary as the protocol executes. We will work in the message passing model, where each pair of processors may communicate by sending messages along channels. It is

⋆ Supported by a National Defense Science and Engineering Graduate Fellowship.

assumed that the channels are reliable, but asynchronous. This means that a message which is sent is eventually received (unaltered), but arbitrarily long delays are allowed. We assume that the sender of a message is always known to the receiver, so the adversary cannot "impersonate" uncorrupted processors.

We will be very conservative in placing limitations on the adversary. We consider the *full information* model, which allows a computationally unbounded adversary who has access to the entire content of all messages as soon as they are sent. We allow the adversary to control message scheduling, meaning that message delays and the order in which messages are received may be maliciously chosen. One may consider a *non-adaptive* adversary, who must fix the t faulty processors at the beginning of the protocol, or an *adaptive* adversary, who may choose the t faulty processors as the protocol executes. Since we are proving an impossibility result, we consider non-adaptive adversaries (this makes our result stronger). We will consider values of t which are $= cn$ for some positive constant $c < \frac{1}{3}$. (The problem is impossible to solve if $t \geq \frac{n}{3}$.) We define the running time of an execution in this model to be the maximum length of any chain of messages (ending once all good processors have decided).

In the asynchronous setting, the seminal work of Fischer, Lynch, and Paterson [10] proved that no deterministic algorithm can solve Byzantine agreement, even for the seemingly benign failure of a single unannounced processor death. More specifically, they showed that any deterministic algorithm may fail to terminate. In light of this, it is natural to consider randomized algorithms with a relaxed termination requirement, such as terminating with probability one. In quick succession following the result of [10], Ben-Or [3] and Bracha [5] each provided randomized algorithms for asynchronous Byzantine agreement terminating with probability one and tolerating up to $t < \frac{n}{5}$ and $t < \frac{n}{3}$ faulty processors respectively. These algorithms feature a relatively simple and intuitive structure, but suffer greatly from inefficiency, as both terminate in expected exponential time. However, when the value of t is very small, namely $\mathcal{O}(\sqrt{n})$, the expected running time is constant.

This state of affairs persisted for a surprising number of years, until the recent work of Kapron, Kempe, King, Saia, and Sanwalani [12] demonstrated that polynomial-time (in fact, polylogarithmic time) solutions are possible. They presented a polylogarithmic-time algorithm tolerating up to $(\frac{1}{3} - \epsilon)n$ faulty processors (for any positive constant ϵ) which is Monte Carlo and succeeds with probability $1 - o(1)$ [13]. The protocol is quite technically intricate and has a complex structure. It subtly combines and adapts several core ingredients: Feige's lightest bin protocol [9], Bracha's exponential time Byzantine agreement protocol (run by small subsets of processors) [5], the layered network structure introduced in [16,17], and averaging samplers.

It seems quite hard to adapt the techniques of Kapron et al. to obtain a Las Vegas algorithm and/or an algorithm against an adaptive adversary, since their protocol relies heavily on universe reduction to ultimately reduce to a very small set of processors. Once we reduce to considering a small subset of the processors, an adaptive adversary could choose to corrupt the entire subset. Even against a non-adaptive adversary, there is always some chance that the small subset we ultimately choose will contain a high percentage of faulty processors. This is essentially why the Kapron et al. protocol incurs a (small) nonzero probability of failure.

Compared to the protocols of Ben-Or [3] and Bracha [5], the Kapron et al. protocol [12,13] appears to be a distant point in what may be a large landscape of possible algorithms. The full range of behaviors and tradeoffs offered by this space remains to be explored. Many interesting questions persist: is there a Las Vegas algorithm that terminates in expected polynomial time? Is there an expected polynomial time algorithm against an adaptive adversary? Is there a much simpler algorithm that performs comparably to the Kapron et al. algorithm, or at least runs in polynomial time with high probability?

In this work, we investigate why simple Las Vegas algorithms in the spirit of [3,5] cannot deliver expected polynomial running time for linear values of t (i.e. $t = cn$ for some positive constant c). More precisely, we define a natural class of protocols which we call *fully symmetric round protocols*. This class encompasses Ben-Or [3] and Bracha's protocols [5], but is considerably more general. Roughly speaking, a protocol belongs to this class if all processors follow the same program proceeding in broadcast rounds where the behavior is invariant under permutations of the identities of the processors attached to the validated messages in each round. In other words, a processor computes its message to broadcast in the next round as a randomized function of the set of messages it has validated, without regard to their senders. We additionally constrain the protocols in the following way. Whenever a processor chooses its message randomly, it must choose from a *constant number* of possibilities. This means that at each step of the protocol, a processor will make a random choice between at most R alternatives, where R is a fixed constant. Note that the set of alternatives itself can vary; it is only the maximum *number* of choices that is fixed. We give a formal description of fully symmetric round protocols in Section 3. We will prove that for any algorithm in this class which solves asynchronous Byzantine agreement, there exists some input values and some adversarial strategy which causes the expected running time to be exponential in n, when $t = cn$ for any fixed positive constant c.

Our general proof strategy is to consider a chain of E-round executions (for some suitably large value E) where the behavior of some good processors is the same between any two adjacent executions in the chain, and the two ends of the chain must have different decision values. This implies that some execution in the chain must not have terminated within E rounds. This is reminiscent of a strategy often used to prove a lower bound of t rounds for deterministic protocols in the synchronous setting (see [8] for example). Employing this sort of strategy for randomized algorithms presents an additional challenge, since any particular execution may be very unlikely. To address this, we consider classes of closely related executions where an adversary is able to exert enough control over a real execution to force it to stay within a chosen class with significant probability.

We view this work not as a primarily negative result, but rather as a guide in the search for new efficient Byzantine agreement algorithms in the asynchronous, full information setting. The goal of this paper is to illuminate some of the obstacles that must be surmounted in order to find an efficient Las Vegas protocol and to spur new thinking about protocols which lie outside the confines of our impossibility result without requiring the full complexity of the Kapron et al. protocol. We hope that the final outcome of this line of research will be interesting new algorithms as well as a greater understanding of the possible features and tradeoffs for protocols in this environment.

Other Related Work In the synchronous, full-information setting, polylogarithmic round randomized protocols for byzantine agreement against a non-adaptive adversary were given by King, Saia, Sanwalani, and Vee [16,17], Ben-Or, Pavlov, and Vaikuntanathan [4], and Goldwasser, Pavlov, and Vaikuntanathan [11]. Restricting the adversary to be non-adaptive is necessary to achieve polylogarithmic time protocols (for values of t which are linear in n), since Bar-Joseph and Ben-Or [2] have proven that any randomized, synchronous protocol against a fail-stop, full information adversary who can adaptively fail t processors must require at least $\frac{t}{\sqrt{n \log n}}$ rounds in expectation. Another lower bound for randomized Byzantine agreement protocols was proven by Attiya and Censor [1], who showed that for each integer k, the probability that a randomized Byzantine agreement algorithm tolerating t faults with n processors does not terminate in $k(n-t)$ steps is at least $1/c^k$ for some constant c. This bound holds even against a considerably weaker adversary than we are considering.

Recent work of King and Saia [14,15] has provided Byzantine agreement protocols in the synchronous setting with reduced communication overhead, namely $\tilde{\mathcal{O}}(n^{3/2})$ bits in the full information model against a non-adaptive adversary [14], and $\tilde{\mathcal{O}}(\sqrt{n})$ bits against an adaptive adversary under the assumption of private channels between all pairs of processors [15]. The use of averaging samplers in recent protocols is foreshadowed by a synchronous protocol presented by Bracha [6] that assigned processors to committees in a non-constructive way. Chor and Dwork [7] provide an excellent survey that covers this as well as the other early work we have referenced.

2 Preliminaries

We begin by formally specifying the model we use and developing a needed mathematical definition.

The Model. We consider n processors who communicate asynchronously by sending and receiving messages. We assume that the communication channel between two processors never alters any messages, and that the sender of a message can always be correctly determined by the receiver. To model asynchrony, we follow the terminology of [10]. We suppose there is a *message buffer*, which contains all messages which have been sent but not yet received. A *configuration* includes the internal states of all processors as well as the contents of the message buffer. A protocol executes in *steps*, where a single step between two configurations consists of a single processor p receiving a value m from the message buffer, performing local computation (which may involve randomness), and sending a finite set of messages to other processors (these are placed in the message buffer). We note that the value returned by the message buffer is either a message previously sent to p or $\emptyset$ (which means that no message is received). The only constraint on the non-deterministic behavior of the message buffer is that if a single processor p takes infinitely many steps, then every message sent to p in the message buffer is eventually received by p.

We suppose there is an *adversary* who controls some t of the processors. We assume these t processors are fixed from the beginning of the protocol. These will be called the *faulty* processors, while the other processors will be called *good* processors. The faulty processors may behave arbitrarily and deviate from the protocol in malicious ways. The adversary also controls the message scheduling (i.e. it decides which processor takes the next step and what the message buffer returns, subject to the constraints mentioned above). Our adversary is computationally unbounded, and has access to the content of all messages as soon as they are sent. Based on this information, the adversary can adaptively decide in what order to deliver messages, subject only to the constraint that all messages which are sent between good processors must eventually be delivered.

We model the use of randomness in a protocol by allowing each processor to sample from its own source of randomness, which is independent of the sources sampled by other processors and *unpredictable* to the adversary. This means that before a good processor samples from its random source, the adversary will know only the distribution of the possible outcomes and nothing more.

Adjusting Probability Distributions. We will constrain our fully symmetric round protocols to always choose the next message randomly via some distribution on at most R possibilities, where R is a fixed constant. We note that the possible messages themselves can change according to the state of the processor as the protocol progresses: it is only the *number* of choices that is constrained, not the choices themselves. Since the probability distributions on R values can be arbitrary, we will define closely related distributions which have more convenient properties for our analysis.

We let $\mathcal{D}$ denote a distribution on a set $\mathcal{S}$ of size at most R. We let ρ_s denote the probability that $\mathcal{D}$ places on $s \in \mathcal{S}$. In our proof, we will be considering t samples of such a distribution $\mathcal{D}$. For each $s \in \mathcal{S}$, the expected number of times that s occurs when t independent samples of $\mathcal{D}$ are taken is $\rho_s t$. In general, this may not be a integer. We will prefer to work with integral expectations, so we define an alternate distribution $\widetilde{\mathcal{D}}$ on the same set $\mathcal{S}$. We let $\tilde{\rho}_s$ denote the probability that $\widetilde{\mathcal{D}}$ places on s for each $s \in S$. The definition of $\widetilde{\mathcal{D}}$ is motivated by two goals: we will ensure that $\tilde{\rho}_s t$ is an positive integer for each $s \in S$, and also that $\tilde{\rho}_s$ and ρ_s are sufficiently close for each $s \in \mathcal{S}$.

Since the size of $\mathcal{S}$ is at most R, there must exist some $s^* \in S$ such that $\rho_{s^*} \geq \frac{1}{R}$. We fix this s^*, and we also fix a small real number $\epsilon > 0$ (whose precise size with respect to t, R will be specified later). For all $s \in \mathcal{S} - \{s^*\}$, we define $\tilde{\rho}_s$ to be the least positive integer multiple of $\frac{1}{t}$ which is $\geq \max\{\rho_s, \epsilon\}$. For s^*, we define $\tilde{\rho}_{s^*} = 1 - \sum_{s \in \mathcal{S}-\{s^*\}} \tilde{\rho}_s$.

Lemma 1. *When $t > R^2$ and $0 < \epsilon < \frac{1}{R^2} - \frac{1}{t}$, $\widetilde{\mathcal{D}}$ is a probability distribution on $\mathcal{S}$, and $\tilde{\rho}_s t$ is a positive integer for each $s \in \mathcal{S}$.*

We will additionally use the following consequence of the Chernoff bound. (The proofs of both lemmas can be found in the full version of this paper.)

Lemma 2. *Let $\mathcal{D}$ be an arbitrary distribution on a set $\mathcal{S}$ of at most R possible values, and let $\widetilde{\mathcal{D}}$ be defined from $\mathcal{D}$ as above, with $t = cn > (\frac{2}{c})R^2$ and $\epsilon < \frac{c}{2R^2} - \frac{1}{t}$ (where c is a positive constant satisfying $0 < c < \frac{1}{3}$). Let $s \in \mathcal{S}$, and let $\rho_s, \tilde{\rho}_s$ denote the probabilities that $\mathcal{D}$ and $\widetilde{\mathcal{D}}$ assign to this value, respectively. Let $X_1, \ldots, X_{(1-c)t}$ denote independent random variables, each equal to 1 with probability ρ_s and equal to 0 with probability $1 - \rho_s$. Then:*

$$\mathbb{P}\left[\sum_{i=1}^{(1-c)t} X_i \geq \tilde{\rho}_s t\right] \leq e^{-\delta c^3 n/(3(1-c))},$$

where δ is defined to be the minimum of ϵ and $\frac{1}{4R}$.

3 Fully Symmetric Round Protocols

We now define the class of fully symmetric round protocols. In these protocols, communication proceeds in rounds. These are similar to the usual notion of rounds in the synchronous setting, but in the asynchronous setting a round may take an arbitrarily long amount of time and different processors may be in different rounds at any given time. Our definition is motivated by the core structure of Bracha's protocol, so we first review this structure. Bracha's protocol relies on two primitives, called Broadcast and Validate. The broadcast primitive allows a processor to send a value to all other processors and enforces that even faulty processors must send the same value to everyone or no value to anyone. The Validate primitive essentially checks that a value received via broadcast could have been sent by a good processor (we elaborate more on this below). Bracha describes the basic form of a round of his protocol as follows[1]:

```
round(k)
    Broadcast(v)
    wait till Validate a set $S$ of $n - t$ $k$-messages
    $v := N(k, S)$
```

Here, a k-message is a message broadcast by a processor in round k, and N is the *protocol function* which determines the next value to be broadcast (N

[1] Bracha refers to this as a "step" [5] and uses the terminology of "round" a bit differently.

is randomized). In Bracha's protocol, N considers only the set of k-messages themselves, and does not consider which processors sent them. This is the "symmetric" quality which we will require from fully symmetric round protocols. This structure and symmetry also characterize Ben-Or's protocol [3], except that the Broadcast and Validate primitives are replaced just by sending and receiving. We will generalize this structure by allowing protocol functions N which consider messages from earlier rounds as well.

Fully symmetric round protocols will invoke two primitives, again called Broadcast and Validate. We assume these two primitives are instantiated by *deterministic* protocols. In each asynchronous round, a processor invokes the Broadcast primitive and broadcasts a message to all other processors (that message will be stamped with the round number). We will describe the properties of the broadcast primitive formally below. To differentiate from the receiving of messages (which simply refers to the event of a message arriving at a processor via the communication network), we say a processor p *accepts* a message m when p decides that m is the outcome of an instantiation of the broadcast primitive. When we refer to the *round number of a message*, we mean the round number attached to the message by its sender. For a fully symmetric round protocol, a round can be described as follows:

```
round(k)
  Broadcast(v)
  wait till Validate a set $S$ of $n-t$ $k$-messages
  let $S'$ denote the set of all validated $i$-messages for all $i<k$
  $v := N(k, S \cup S')$
```

The message to be broadcast in the first round is computed as $N(0, b)$, where b is the input bit of the processor. As in the case of Bracha's protocol, we consider the set of messages $S \cup S'$ as divorced from the sender identities, so the protocol function N does not consider which processor sent which message. Note here that we have allowed the protocol function to consider all currently validated messages with round numbers $\leq k$ (i.e. were broadcast by their senders in rounds $\leq k$). In contrast, the Validate algorithm may consider the processor identities attached to messages.

In summary, in each round a processor waits to validate $n-t$ messages from other processors for that round. Once this occurs, it applies the protocol function N to the set of validated messages. This protocol function determines whether or not the processor decides on a final bit value at this point (we assume this choice is made deterministically), and also determines the message to be broadcast in the next round. This choice may be made randomly. We note that choice of whether to decide a final bit value (and what that value is) only depends on the set of accepted messages themselves, and does not refer to the senders.

Key Constraint on Randomized Behavior. We constrain a processor's random choices in the following crucial way. We assume that when a processor employs randomness to choose its message to broadcast in the next round, it chooses from at most R possibilities, where R is a fixed, global constant independent of all other parameters (e.g. it does not depend on the round number or the total number of processors)[2]. Note that the choices themselves may depend on the round number, the total number of processors, etc. The messages themselves may also be quite long - there is no constraint on their bit length.

Full Symmetry. Fully symmetric round protocols are invariant under permutations of the identities associated with validated messages in each round. At the end of each round, a good processor may consult all previously validated messages (divorced from any information about their senders) and must choose a new message to broadcast at the beginning of the next round. It may make this choice randomly, so we think of the set $S \cup S'$ of all previously validated messages as determining a distribution on a *constant* number of possible messages for the next round. We emphasize that since $S \cup S'$ is just the set of the bare messages themselves, it also contains no information about which messages were sent by the same processors, so the distribution determined by $S \cup S'$ is invariant under all permutations of the processor identities associated with messages for each round where the permutations may *differ* per round.

Broadcast and Validate Primitives. We now formally define the properties we will assume for the broadcast and validate primitives. We recall that these are assumed to be deterministic. We first consider broadcast. We suppose that the broadcast primitive is invoked by a processor p in order to send message m to all other processors. We consider the n processors as being numbered 1 through n, which allows us to identify the set of processors with the set $[n] := \{1, 2, \ldots, n\}$. We will assume that for each permutation π of the set of $[n]$, there exists a finite schedule of events that can be applied (starting from the current configuration) such that at the end of the sequence of events, all processors have accepted the message m and that for each i from 1 to $n-1$, there is a prefix of the schedule such that at the end of the prefix, exactly the processors $\pi(1), \ldots, \pi(i)$ have accepted the message m. Essentially, this means that every possible order of acceptances can be achieved by some applicable schedule. (Note that within these schedules, all processors act according to the protocol.) More formally, we make the following definition:

Definition 1. *We say a broadcast protocol allows* ***arbitrary receiver order*** *if for any processor p invoking the protocol to broadcast a message m and for any permutation π of $[n]$, there exists a finite schedule σ_π of events that can be applied consecutively starting from the initial configuration such that there exist prefixes $\sigma_1, \ldots, \sigma_n = \sigma_\pi$ of σ_π such that in the configuration resulting from σ_i, exactly the processors $\pi(1), \ldots, \pi(i)$ have accepted m, and no other processors have.*

It is clear that this property holds if one implements broadcast simply by invoking the send and receive operations on the communication network. This property also holds for Bracha's broadcast primitive, which enforces that even faulty

[2] This constraint is satisfied by Ben-Or and Bracha's protocols, since both choose from two values whenever they choose randomly.

processors must send the same message to all good processors or no message at all. This property will be useful to our adversary (who controls scheduling) because it allows complete control over the order in which processors accept messages. We assume that our fully symmetric round protocols treat each invocation of the broadcast primitive as "separate" from the rest of the protocol in the sense that any messages sent not belonging to an instance of the broadcast primitive do not affect a processor's behavior within this instance of the broadcast primitive.

We consider the Validate primitive as an algorithm V which takes as input the set of all accepted messages so far along with accompanying information specifying the sender of each message. The algorithm then deterministically proceeds to mark some subset of the previously accepted messages as "validated". We assume that this algorithm is monotone in the following sense. We let $W^+ \subseteq S^+$ be two sets of accepted message and sender identity pairs (we use the $^+$ symbol to differentiate these sets of messages with senders from sets of messages without sender identity attached). Then if a message, sender pair $(m, p) \in W^+$ is marked valid by $V(W^+)$, then this same pair (m, p) will be marked valid by $V(S^+)$ as well. In other words, marking a message as valid is a decision that cannot be reversed after new messages are accepted.

We assume the validation algorithm is called each time a new message is accepted to check if any new messages can now be validated. Bracha's Validate algorithm is designed to validate only messages that could have been sent by good processors in each round. It operates by validating an accepted message m for round k if and only if there are $n-t$ validated messages for round $k-1$ that could have caused a good processor to send m in round k (i.e. m is an output of the protocol function N that occurs with nonzero probability when these $n-t$ validated round $k-1$ messages are used as the input set). In the context of Bracha's algorithm, where the behavior for one round only depends on the messages from the previous round, this essentially requires faulty processors to "conform with the underlying protocol" [5] (up to choosing their supposedly random values maliciously) or have their messages be ignored.

In the context of protocols that potentially consider messages from *all* previous rounds, one might use a stronger standard for validation. For instance, to validate a message m_k for round k sent by a processor p, one might require that there are messages $m_1, \ldots, m_{k-1}$ for rounds 1 through $k-1$ sent by p which are validated and that there are sets of validated messages $S_1, \ldots, S_{k-1}$ such that each S_i contains messages for rounds $\leq i$ and exactly $n-t$ messages for round i, $S_i \subset S_{i+1}$ for each $i < k-1$, and $N(i, S_i) = m_{i+1}$ with non-zero probability for each i from 1 to $k-1$. This essentially checks that there is a sequence of sets of validated messages that p could have considered in each previous round that would have caused a good processor to output messages $m_1, \ldots, m_k$ in rounds 1 through k with non-zero probability.

Roughly, we will allow all validation algorithms that never fail to validate a message m sent by a good processor p when all of the previous messages sent by p and all of the messages that caused the processor p to send the message m

have been accepted. We call such validation algorithms *good message complete.* We define this formally as follows.

Definition 2. *A Validate algorithm V is* **good message complete** *if the following condition always holds. Suppose that $S_1^+ \subset S_2^+ \subset \ldots \subset S_k^+$ are sets of validated messages (with sender identities attached) such that each S_i^+ contains exactly $n-t$ round i messages and $m_1, \ldots, m_k$ occur with non-zero probability as outcomes of $N(1, S_1)$, ..., $N(k, S_k)$ respectively. Then if a set W^+ of messages (with sender identities attached) includes S_k^+ as well as the messages $m_1, \ldots, m_k$ from the same sender, then $V(W^+)$ marks m_k as validated.*

This means that if during a real execution of the protocol, a good processor p computes its first k messages $m_1, \ldots, m_k$ by applying $N(1, S_1)$, $N(2, S_2)$, ..., $N(k, S_k)$ respectively and another good processor q has accepted $m_1, \ldots, m_k$ from p as well as all of the messages in S_k, then q will validate m_k.

We note that the use of Validate protocols which are *not* good message complete seems quite plausible in the Monte Carlo setting at least, since a Monte Carlo algorithm can afford to take some small chance of not validating a message sent by a correct processor. In this case, it would also be plausible to consider randomized validation protocols. However, since we are considering Las Vegas algorithms, we will restrict our attention in this paper to deterministic, good message complete validation protocols.

We have now completed our description of fully symmetric round protocols. In summary, they are round protocols that invoke a broadcast primitive allowing arbitrary receiver order, invoke a Validate primitive that is good message complete, are invariant under permutations of the processor identities attached to the messages in each round, and always make random choices from a constant number of possibilities.

4 Impossibility of Polynomial Time for Fully Symmetric Round Protocols

We are now ready to state and prove our result.

Theorem 1. *For any fully symmetric round protocol solving asynchronous Byzantine agreement with n processors for up to $t = cn$ faults, for $c > 0$ a positive constant, there exist some values of the input bits and some adversarial strategy resulting in expected running time that is exponential in n.*

Proof. We suppose we have a fully symmetric round protocol with resilience $t = cn$. We will assume that t divides n for convenience, and also that ct is an integer. These assumptions will make our analysis a little cleaner, but could easily be removed. We let R denote the constant bound on the number of possibilities for each random choice. E will denote a positive integer, the value of which will be specified later. (It will be chosen as a suitable function of c, R and n, and will be exponential in n when c, R are positive constants.)

We will be considering partial executions of the protocol lasting for E rounds. For convenience, we think of our protocol as continuing for E rounds even if all good processors have already decided (this can be artificially achieved by having decided processors send default messages instead of terminating). The adversary must fix t faulty processors at the beginning of an execution. Once these processors are fixed, we divide the n processors into disjoint groups of t processors each, so there are $\frac{n}{t}$ groups. We will refer to the groups as G_1 through $G_{n/t}$. We choose our groups so that exactly ct of the processors in each group are faulty. The main idea of our proof is as follows. Since the broadcast and validation primitives essentially constrain the behavior of faulty processors, we think of the adversary as controlling only the (supposedly) random choices of faulty processors as well as the message scheduling. This means that when faulty players invoke the randomized function $N(i, S)$, they may maliciously chose any output that occurs with nonzero probability. In all other respects, they will follow the protocol.

The adversary will choose the message scheduling so that the t processors in a single group will proceed in lockstep: the sets of messages that they use as input to N will always be the same in each round. This means that all t processors in a group will be choosing their next message from the *same distribution*. Since there are only a constant number of possibilities and the adversary controls a constant fraction of the processors in the group, it can ensure with high probability that the collection of messages which are actually chosen is precisely equal to the expectation under the adjusted distribution. More precisely, we let $\mathcal{D}$ denote the distribution (on possible next round messages) resulting from applying N to a particular set of messages in a particular round, and we let $\mathcal{S}$ denote the set of (at most R) outputs that occur with nonzero probability. We define $\widetilde{D}$ with respect to $\mathcal{D}$ as in Section 2. Then, with high probability, once the adversary sees the outputs chosen by the $(1-c)t$ good processors in the group, it can choose the messages of the ct faulty processors in the group so that the total number of processors in the group choosing each $s \in \mathcal{S}$ is exactly $\tilde{\rho}_s t$. (This is proven via Lemma 2.)

We can then consider classes of executions which proceed with these groups in lockstep as being defined by the set of messages used as input to N in each round by each group (as well as the sets of sender, round number pairs for the messages, but these pairs are divorced from the messages themselves). With reasonable probability, an adversary who controls the message scheduling and the random choices of faulty processors can force a real execution to stay within such a class for E rounds. We will prove there exists such a class in which some good processors fail to decide in the first E rounds. Putting this all together, we will conclude that there exist some values of the input bits and some adversarial strategy that will result in expected running time that is exponential in n.

Our formal proof begins with the following definition.

Definition 3. *An E-round lockstep execution class $\mathcal{C}$ is defined by a setting of the input bits for each group (processors in the same group will have the same input bit), message sets S_i^j for all $1 \leq i \leq E$ and $1 \leq j \leq n/t$ where S_i^j is used* *as the input to N in round i by each processor in group G_j during some real execution, and sets Z_i^j of processor, round number pairs consisting of all pairs (p,k) such that the message broadcast by processor p in round k is contained in S_i^j. We require that for all i,j, if $(p,k) \in Z_i^j$ and processors p and p' are in the same group $G_{j'}$, then $(p',k) \in Z_i^j$ as well.*

We have required that an E-round lockstep execution class $\mathcal{C}$ describe *some real execution*, but note that such an execution is not unique. There are many possible executions that correspond to the same class $\mathcal{C}$. It is crucial to note that the messages in sets S_i^j are not linked to their sender, round number pairs in Z_i^j. In other words, given the sets Z_i^j and S_i^j, one has cumulative information about the messages and the senders, but there is no specification of who sent what.

We will construct a chain $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_L$ of E-round lockstep execution classes with the following properties:

1. For each S_i^j in each $\mathcal{C}_\ell$, the number of occurrences of each message s is exactly equal to $\tilde{\rho}_s t$, where $\tilde{\rho}_s$ denotes the probability on s in the distribution $\tilde{\mathcal{D}}$ defined from $\mathcal{D}$ as in Section 2, where $\mathcal{D}$ is the distribution on possible messages induced by $N(i-1, S_{i-1}^j)$.
2. Each $\mathcal{C}_\ell$ and $\mathcal{C}_{\ell+1}$ differ only in the sets S_i^j for *one* group G_j.
3. It is impossible for any good processor to decide the value 1 during an execution in class $\mathcal{C}_0$.
4. It is impossible for any good processor to decide the value 0 during an execution in class $\mathcal{C}_L$.

Once we have such a chain of E-round lockstep execution classes, we may argue as follows. Since each $\mathcal{C}_\ell$ and $\mathcal{C}_{\ell+1}$ differ only in the behavior of processors in a single group, it is impossible for all good processors to have decided 0 in an execution in class $\mathcal{C}_\ell$ and all good processors to have decided 1 in an execution in class $\mathcal{C}_{\ell+1}$. Since the only decision value possible in $\mathcal{C}_0$ is 0 and the only decision value possible in $\mathcal{C}_L$ is 1, there must be some $\mathcal{C}_{\ell^*}$ which leaves some good processors undecided. In other words, any execution in this class C_{ℓ^*} does not terminate in $\leq E$ rounds. Finally, we will show that when the input bits match the inputs for $\mathcal{C}_{\ell^*}$, the adversary can (with some reasonable probability) cause a real execution to fall in class $\mathcal{C}_{\ell^*}$. Since E is exponential in n whenever R, c are positive constants, this will prove that the expected running time in this case is exponential.

To generate the chain of E-round lockstep execution classes with the properties required above, we first observe that given settings of the input bits for each group and sets $z_i^j \subseteq [n]$ of size $n-t$ (each is a complement of some group) for every i from 1 to E and every j from 1 to n/t, we can create a real execution in which the $n-t$ round i round messages validated by group G_j in round i are exactly those sent by senders in the set z_i^j. We then create C_0 by using arbitrary sets z_i^j and input bits all equal to zero. We then employ a recursive algorithm designed to gradually shift the input bits to 1 by first making incremental changes to the sets z_i^j so that a particular group will not be heard by the other groups, making it “safe” to change the inputs for that group without affecting other processors.

To reach an E-round lockstep execution class where a particular group's messages are not heard by other processors, we follow an inductive strategy with round E acting as the base case. Suppose that we want to change the inputs for processors in group G_j. We cannot do this immediately if it might affect the behavior of processors outside this group in the first E rounds. We define $i-1$ to be the *earliest* round in which the set $z_{i-1}^{j'}$ for some other group $G_{j'}$ includes a sender in G_j. We now seek to change the set $z_{i-1}^{j'}$ to be the complement of group G_j. Now we have a new instance of the same problem: in order to change what messages group $G_{j'}$ members accept in round $i-1$ without affecting processors outside of this group, we must first get to a lockstep execution class where processors outside of group $G_{j'}$ do not accept messages sent from group $G_{j'}$ with round numbers $\geq i$ until they have completed E rounds. The important thing to notice here is that the new instance of the problem always involves a *higher* round number. Hence, we can formulate this as a recursion, and eventually we reach a point where it is enough to ensure that the messages of some group $G_{j''}$ with round numbers $\geq i'$ are not heard by some other group $G_{j'''}$ in round E. This is now easy to do, since we can arrange for the $n-t$ other round E messages to be validated while we delay the messages with round numbers $\geq i'$ from group $G_{j''}$ to $G_{j'''}$, so the processors in group $G_{j'''}$ can exit round E. (Notice here that E will be the earliest round in which any group may receive messages with round number $\geq i'$ from $G_{j''}$, and this ensures that these messages cannot be needed to validate the round E messages of processors outside $G_{j''}$.) More intuition about how this occurs as well as the formal algorithm and proof can be found in the full version of this paper.

4.1 Completing the Proof of Theorem 1

We consider our chain of E-round lockstep execution classes $\mathcal{C}_0, \ldots, \mathcal{C}_L$ satisfying properties 1 through 4. Among these, there is some $\mathcal{C}_{\ell^*}$ which results in some good processors remaining undecided after E rounds. We now use Lemma 2 to complete our proof. We recall that this lemma shows that when the good processors in a group each sample their next message independently from the same distribution $\mathcal{D}$ on at most R possibilities, with high probability the adversary can choose the “random” bits of the faulty processors in the group to ensure that the number of times each possible message is chosen within the group exactly matches the expected number under distribution $\widetilde{\mathcal{D}}$.

We let δ' denote the value $\delta c^3/(3(1-c))$ appearing in the statement of Lemma 2. We note that δ' is a positive constant which can be chosen to depend only on R and c (recall the ϵ is chosen with respect to R and c). We consider an execution which begins with the same input bits as $\mathcal{C}_{\ell^*}$. As the execution runs, the adversary will choose the message scheduling and the supposedly random bits for the faulty processors in an attempt to create message sets through the first E rounds that match the sets S_i^j associated to $\mathcal{C}_{\ell^*}$. For details on how the message scheduling is chosen, see the full version of this paper.

In order for the adversary to be successful in creating an execution that falls into class C_{ℓ^*}, it must ensure that the messages chosen in each round by each group conform precisely to the expected numbers for each possibility under the corresponding distribution $\widetilde{\mathcal{D}}$. This can be done as long as the number of good processors in the group choosing each possibility s do not exceed the expected number, $\tilde{\rho_s}t$. When this occurs, the adversary can set the messages of the faulty processors in the group so that each expectation is matched precisely. We note that the sets S_i^j always contain *all* of the round k messages sent by a group or none of them (recall we have required that if $(p,k) \in Z_i^j$ for any processor p, any rounds i,k, and any group j, then $(p',k) \in Z_i^j$ for all processors p' in the same group as p). Thus, as long as the adversary achieves the desired multi-set of messages for each group, the sets S_i^j of $\mathcal{C}_{\ell^*}$ will be attained. (It does not matter which processor from each group sends which message, as long as the multi-set of messages produced by each group matches the specification of $\mathcal{C}_{\ell^*}$.)

Since there are at most R possible messages for each group and there are $n/t = 1/c$ groups, the union bound in combination with Lemma 2 ensures that the probability of the adversary failing in any given round is at most $\frac{R}{c}e^{-\delta' n}$. Thus, the adversary will succeed in producing the sets S_i^j associated with $\mathcal{C}_{\ell^*}$ through E rounds with probability at least $1 - \frac{ER}{c}e^{-\delta' n}$. When the adversary succeeds, some good processors will remain undecided at the end of E rounds.

We now fix the value of E as $E := \frac{c}{2R}e^{\delta' n}$. This is exponential in n, and the probability that the adversary can force the execution to last for at least E rounds is $\geq \frac{1}{2}$. This proves that the expected running time is exponential. This completes our proof of Theorem 1.

5 Directions for Future Work

We hope that the restrictions we placed on fully symmetric round protocols in order to implement our proof strategy may provide useful clues for where one should look when searching for polynomial expected time algorithms (particularly Las Vegas algorithms). Informally speaking, we may ask: how far does one have to go beyond the realm of fully symmetric round protocols in order to find an expected polynomial time algorithm? Does one have to abandon symmetry completely? Or might one deviate from our specifications in more subtle ways?

Weaker Symmetry. For instance, we could consider an enlarged class of protocols that is symmetric in a weaker sense: behavior could still be invariant under permutations of the processor identities attached to accepted messages, but these permutations could be fixed for the entire history of previous rounds, instead of allowed to change per round. We do not know whether our impossibility result can be extended to protocols exhibiting this weaker kind of symmetry.

More Randomness. It is also intriguing to consider the small change of lifting the restriction on the number of random choices. Though our probabilistic analysis is not nearly optimized, it does seem fairly sensitive to the number of possibilities considered when a processor makes a random choice. Having more choices will considerably decrease the adversary's chances of arranging the numbers of all outcomes to conform with their adjusted expectations. However, it is not clear how to leverage using more randomness to achieve faster Las Vegas algorithms.

References

1. Attiya, H., Censor, K.: Lower bounds for randomized consensus under a weak adversary. In: PODC, pp. 315–324 (2008)
2. Bar-Joseph, Z., Ben-Or, M.: A tight lower bound for randomized synchronous consensus. In: PODC, pp. 193–199 (1998)
3. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols. In: PODC, pp. 27–30 (1983)
4. Ben-Or, M., Pavlov, E., Vaikuntanathan, V.: Byzantine agreement in the full-information model in o(log n) rounds. In: STOC, pp. 179–186 (2006)
5. Bracha, G.: An asynchronous [(n-1)/3]-resilient consensus protocol. In: PODC, pp. 154–162 (1984)
6. Bracha, G.: An O(lg n) expected rounds randomized byzantine generals protocol. In: STOC, pp. 316–326 (1985)
7. Chor, B., Dwork, C.: Randomization in Byzantine Agreement. In: Advances in Computing Research, vol. 5, pp. 443–497. JAI Press, Greenwich (1989)
8. Dolev, D., Strong, R.: Polynomial algorithms for byzatine agreement. In: STOC, pp. 401–407 (1982)
9. Feige, U.: Noncryptographic selection protocols. In: FOCS, pp. 142–153 (1999)
10. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. In: PODS, pp. 1–7 (1983)
11. Goldwasser, S., Pavlov, E., Vaikuntanathan, V.: Fault-tolerant distributed computing in full-information networks. In: FOCS, pp. 15–26 (2006)
12. Kapron, B.M., Kempe, D., King, V., Saia, J., Sanwalani, V.: Fast asynchronous byzantine agreement and leader election with full information. In: SODA, pp. 1038–1047 (2008)
13. Kapron, B.M., Kempe, D., King, V., Saia, J., Sanwalani, V.: Fast asynchronous byzantine agreement and leader election with full information. ACM Transactions on Algorithms 6(4) (2010)
14. 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)
15. King, V., Saia, J.: Breaking the O(n^2) bit barrier: scalable byzantine agreement with an adaptive adversary. In: PODC, pp. 420–429 (2010)
16. King, V., Saia, J., Sanwalani, V., Vee, E.: Scalable leader election. In: SODA, pp. 990–999 (2006)
17. King, V., Saia, J., Sanwalani, V., Vee, E.: Towards secure and scalable computation in peer-to-peer networks. In: FOCS, pp. 87–98 (2006)
18. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27(2), 228–234 (1980)

Randomized Consensus in Expected $O(n^2)$ Total Work Using Single-Writer Registers

James Aspnes*

Yale University

Abstract. A new weak shared coin protocol yields a randomized wait-free shared-memory consensus protocol that uses an optimal $O(n^2)$ expected total work with single-writer registers despite asynchrony and process crashes. Previously, no protocol was known that achieved this bound without using multi-writer registers.

1 Introduction

The **consensus** problem is to get a group of n processes to agree on a bit. In a **wait-free randomized consensus** protocol, each process starts with an input bit and produces a decision bit; the protocol is correct if it satisfies **agreement**, where all processes that finish the protocol choose the same decision bit; **validity**, where every decision bit is equal to some process's input; and **probabilistic termination**, where every non-faulty process completes the protocol after a finite number of its own steps with probability 1. These conditions are required to hold despite asynchrony and up to $n-1$ crash failures.

We consider consensus in an asynchronous shared-memory system where the timing of events and failures is under the control of an **adaptive adversary**, one that can observe the complete state of the system—including the internal states of process—but that cannot predict future coin flips made by the processes.

Processes communicate by reading and writing a collection of **atomic registers**, which may be **single-writer** (only one particular process is allowed to write to each register) or **multi-writer** (any process can write to any register). In either case we assume that a register can be read by all processes. The cost of a protocol is measured by counting either the expected total number of operations carried out by all processes (the **total work** or **total step complexity**) or the maximum expected number of operations carried out by any single process (the **individual work** or **individual step complexity**). For either measure, a worst-case adversary is assumed.

Single-writer registers were used in most early shared-memory consensus protocols [1, 3, 6, 8, 11–14, 18]. More recent protocols [4, 7] have used multi-writer registers for increased efficiency. Though multi-writer registers can be implemented from single-writer registers (see, for example, [9, Theorem 10.9]), this imposes a linear blow-up in the costs, and the question of whether a stronger

* Supported in part by NSF grant CCF-0916389.

lower bound might apply to single-writer register protocols than to multi-writer register protocols has remained open [7]. We answer this question, by giving a wait-free randomized consensus protocol using only single-writer registers that matches the lower bound on total step complexity for multi-writer registers.

1.1 Prior Work

Wait-free randomized consensus can be solved using shared memory by reduction to a **weak shared coin** [5]. A weak shared coin is a protocol in which each process chooses a bit, with the property that there is some **agreement parameter** $\delta > 0$ such that for each possible value b, all processes choose b with probability at least δ for any adversary strategy. A construction of Aspnes and Herlihy [5], which uses only single-writer registers, shows that any weak shared coin protocol with expected total work $T(n)$ and agreement parameter δ can be used to solve consensus with $O\left(\left(n^2 + T\left(n\right)\right)/\delta\right)$ expected total work. In particular, finding a weak shared coin with total work $O(n^2)$ and constant agreement parameter gives consensus in $O(n^2)$ expected total work.

Though early randomized consensus protocols [1, 13] did not use a shared coin, much of the subsequent development of randomized consensus protocols for the adaptive-adversary model has been based on this technique [3, 4, 6–8, 11, 12, 14, 18]. The typical structure of a shared coin protocol is to have the processes collectively generate $\Theta(n^2)$ random ± 1 votes, and have each process choose what it sees as the majority value. The intuition is that after $\Theta(n^2)$ votes, the margin of the majority value is likely to be $\Omega(n)$, large enough that the adversary cannot disguise it by selectively delaying processes casting votes it doesn't like.

The main variation between protocols is in how they detect when enough votes have been cast. For many years, the best known protocol was that of Bracha and Rachman [12]. In this protocol, each process maintains in its own register both a count of how many votes it has generated so far and the sum of all of its votes. After every $n/\log n$ votes, a process collects all of the register values (by reading all n registers), and decides on the majority value if the sum of the counts exceeds n^2. The dominant cost is the cost of the collect, which amortizes at $O(\log n)$ register operations for each of the $O(n^2)$ total votes, giving $O(n^2 \log n)$ total work.

The reason for checking the vote count every $n/\log n$ steps is that it guarantees that at most $n^2/\log n$ **extra votes** can be generated once the initial n^2 **common votes** are cast. It can then be shown that (a) the common votes produce with constant probability a net majority at any fixed multiple of n, their standard deviation; while (b) using Hoeffding's inequality, the net extra votes seen by any one process have probability less than $\frac{1}{n^2}$ of exceeding $2n$, giving a low probability that any of the processes sees a large shift from the extra votes. Factoring in the additional shift of up to $n-1$ votes that have been generated but not written still leaves a constant probability that all processes see the same majority value.

The main excess cost in the Bracha-Rachman protocol is the $\Theta(\log n)$ amortized cost per vote of doing a full collect every $n/\log n$ votes. This is needed to keep the extra votes from diverging too much: if we collect only every $\Theta(n)$ votes, we would expect a constant probability for each process that the $\Theta(n^2)$ extra votes it sees change the majority value.

The goal of finding an $O(n^2)$ total-work shared coin with constant agreement parameter was finally achieved by Attiya and Censor using multi-writer registers [7] in a paper that also showed that $\Omega(n^2)$ total work was necessary for consensus. The key idea is to use Bracha-Rachman, modified to do collects every n votes, but add a single **termination bit** that shuts down voting immediately once some process detects termination. This makes all processes see essentially the same set of extra votes, meaning that it is no longer necessary to bound the effect of the extra votes separately for each process. However, a single multi-writer register is needed to implement the termination bit, even though the rest of the protocol uses only single-writer registers.

1.2 Our Approach

We show that the multi-writer bit is not needed: it can be replaced by an array of single-writer termination bits (one for each process) that propagate via a gossip protocol running in parallel with the voting mechanism at an amortized cost of $O(1)$ operations per vote. The intuition for why this works is that, as more processes detect termination, fewer processes are left voting. So while some processes may see a full $O(n^2)$ extra votes, later processes will see fewer; thus the probability that each process sees enough extra votes to shift the majority drops geometrically, giving a constant total probability that any process returns the wrong value. To make this work, a counting argument is used to show that the k-th process to detect termination sees at most $2n^2/k$ extra votes, and that this bound holds *simultaneously* for all k with constant probability. This avoids a blow-up that would otherwise occur using simple union bounds.

2 The Shared Coin Protocol

Code for the shared coin algorithm is given in Algorithm 1. The structure is similar to the protocol of Bracha and Rachman [12] as improved by Attiya and Censor [7], with the single termination bit of the Attiya-Censor protocol replaced by an array of termination bits that are sampled randomly. The essential idea is that a process repeatedly generates random ± 1 votes (using the `CoinFlip()` subroutine, which generates each value ± 1 with equal probability). These are added to the process's sum field in $a[\mathsf{pid}]$, while at the same time the number of votes the process has generated is written to the count field. Each process checks after every n votes to see if the sum of all the count fields exceeds a threshold $T = 64n^2$, and probes a random termination bit $\mathsf{done}[r]$ before every vote. If either enough votes have been generated or $\mathsf{done}[r]$ is set, the process exits the loop immediately, setting $\mathsf{done}[\mathsf{pid}]$ and returning the sign of the sum of all the sum fields.

```
   shared data:
      Register $a[p]$ for each process $p$, with fields $a[p]$.count and $a[p]$.sum, both
   initially 0.
      Boolean register done[$p$] for each process $p$, initially false.
 1 for $i \leftarrow 1 \ldots \infty$ do
 2     if $i \bmod n = 0$ then
 3         if $\sum_{p=1}^{n} a[p]$.count $\geq T$ then break
 4     end if
 5     Choose $r$ uniformly at random from $\{1 \ldots n\} \setminus \{$pid$\}$.
 6     if done[$r$] = true then break
 7     $v \leftarrow$ CoinFlip()
 8     $a$[pid] $\leftarrow \langle a$[pid].count $+ 1, a$[pid].sum $+ v\rangle$.
 9 end for
10 done[pid] $\leftarrow$ true
11 return sgn $\left(\sum_{p=1}^{n} a[p].\text{sum}\right)$
```

Algorithm 1. Shared coin protocol

3 Analysis

For the analysis of the protocol, we fix an **adversary strategy**. This is a function that selects, after each initial prefix of the computation, which process will execute the next operation, leaving the results of the calls to `CoinFlip` and the random choices of done-bit probes as the only source of nondeterminism. We then wish to show that each outcome ± 1 is chosen by all processes with at least constant probability.

The essential idea is to show first that the sum of the votes generated before each process detects termination is likely to be large and then that the sums of the extra votes seen by each process p are likely to be small simultaneously for all p. In this case, no process's extra votes causes it to see a majority different from the common majority. This approach follows similar arguments used for previous protocols based on Bracha-Rachman [4, 6, 7, 12]. The main new wrinkle is that we consider non-uniform error probabilities, where a process that sets done[p] early is more likely to return the wrong value than a process that sets it later.

We write that a process's i-th vote is **generated** when the process executes the call to `CoinFlip()` in Line 1. We will be more interested in when a vote is generated than when it is ultimately added to $a[p]$. We let $X_1, X_2, X_3, \ldots$ be random variables, with X_t representing the return value of the t-th call to `CoinFlip()` by any process, and let $S_t = \sum_{i=1}^{t} X_i$ be the sum of the first t generated votes. Because each X_i has expectation zero, the sequence $\{S_t\}$ is a martingale, and we can use tools for bounding the evolution of martingales to characterize the total vote trajectory over time.[1]

[1] A good general reference on martingales (and other stochastic processes) is [15]. Discussion of applications of martingales to analysis of algorithms can be found in [2, 16, 17].

Because we are examining votes when they are generated and not when they are written, we say that a process p **observes** a vote X_t generated by q if X_t is generated before p reads $a[q]$ during its final collect. The observed votes are not necessarily read by p or included in its final tally; but since at most one vote generated by q can be missing when p reads $a[q]$, the sum that p computes will differ from the sum it "observes" by at most $n - 1$.

3.1 Overview of the Proof

We now give an outline of the structure of the proof:

1. We bound how far the values in the registers lag behind the generated votes (Lemma 1). This bound is used first to bound the total number of votes generated (in Lemma 2) and later to bound the gap between the generated votes observed by a process and the sum computed by that process during its final collect.
2. We show that the sum S_T of the first T votes is at least $8n$ with probability at least $1/8$ (Lemma 3). This $8n$ majority is our budget for losses in the later stages of the protocol.
3. If the preceding event holds, the probability that S_t ever drops below $4n$ during subsequent votes is shown to be less than $1/8$ (Lemma 4). This bound is obtained by combining Kolmogorov's inequality and the bound on the total number of votes from Lemma 2. A consequence is that it is likely that, for every process, S_t is above $4n$ when the process detects termination and begins its final collect.
4. While a process's final collect is in progress, extra votes may come in that reduce the value seen by the process below $4n$ (this does not contradict Lemma 4, because the adversary may be selective in when it allows the process to read particular registers). While different processes may see different numbers of extra votes (processes that detect termination later will have observe fewer other processes still generating votes), we can bound simultaneously the number of extra votes seen by each process as a function of the order in which they set their termination bit (Lemma 5).
5. The extra votes observed by each process also form a martingale, and thus the probability that they reduce the total by $3n$ or more can be bounded using Azuma's inequality [10]. Even though the different number of extra votes observed by each process varies, the resulting probability bounds form a geometric series that sums to less than $1/8$ (Lemma 6).
6. After subtracting the $3n$ bound on extra votes from the $4n$ bound of the previous step, we have a constant probability that the number of votes observed by every process is at least n. Since the actual total read by each process differs from the observed value by at most $n-1$, this gives that every process sees at least $+1$ votes, proving agreement (Theorem 1).

The choice of the thresholds $8n$ and $4n$ is somewhat arbitrary; these particular values happen to be convenient for the proof, but it is likely that further optimization is possible. The threshold n in the last step is needed because of the gap between generated votes and the values actually stored in the registers.

We now proceed with the details of the proof.

3.2 Deterministic Bounds on Error and Running Time

The following lemma bounds the difference between the generated votes and the values in the registers:

Lemma 1. *Let $\gamma_{pi} = 1$ if vote X_i is generated by p, and 0 otherwise. In any state of the protocol after exactly t calls to* `CoinFlip()` *have been made, we have:*

1. $t - n \le \sum_{p=1}^{n} a[p].\mathsf{count} \le t$.
2. *For all p,* $\left| \left(\sum_{i=1}^{t} \gamma_{pi} X_i \right) - a[p].\mathsf{sum} \right| \le 1$.

Proof. Immediate from inspection of the protocol and the observation that for each process, there can be at most one vote that has been generated in Line 1 but not yet written in Line 1.

Lemma 2. *The total number of votes τ generated during any execution of the protocol is at least T and at most $T + n^2 + n$.*

Proof. Before a process p can finish, it must first write $\mathsf{done}[p]$ in Line 1. Consider the first process to do so. This process can only exit the loop after seeing $\sum_{p=1}^{n} a[p].\mathsf{count} \ge T$, which occurs only if at least T votes have already been generated (and written).

For the upper bound, suppose that at some time, $T + n$ votes have been generated; then from Lemma 1 we have $\sum_{p=1}^{n} a[p].\mathsf{count} \ge T$. Each process can generate at most n more votes before executing the test in Line 1 and exiting. Adding in these votes, summed over all processes, gives a total of at most $(T + n) + n^2 = T + n^2 + n$ votes.

It is easy to see from Lemma 2 that the total work for Algorithm 1 is $O(n^2)$; the main loop contributes an amortized $O(1)$ operations per vote (plus an extra $O(n)$ operations per process for the first execution of the collect in Line 1), while the assignment to $\mathsf{done}[\mathsf{pid}]$ and the final collect contributes $O(n)$ more operations per process. Summing these contributions gives the claimed bound.

3.3 Common Votes and Extra Votes

For each process p, let κ_p be the number of votes generated before p either observes a total number of votes greater than or equal to T in Line 1 or observes a non-**false** value in $\mathsf{done}[r]$ in Line 1. For each t, let ξ_{pt} be the indicator variable for the event that both $t > \kappa_p$ and a vote X_t is generated by some process q before p reads $a[q]$ in Line 1; formally, $\xi_{pt} = [t > \kappa_p] \wedge \gamma_{pt}$, where γ_{pt} is the random variable defined in Lemma 1. Since all votes $X_1 \ldots X_{\kappa_p}$ are generated before p executes Line 1, the sum over all q of votes generated by q before p reads $a[q]$ is given by $S_{\kappa_p} + \sum_t \xi_{pt} X_t$. We will refer to these votes as the votes

observed by p, even though (by Lemma 1) up to one such vote for each process q may not be written to $a[q]$ before p reads it.

We will show that, with constant probability, the total votes observed by every process is bounded away from 0 by at least n in the same direction. The essentially idea is to show that $\left|S_{\kappa_p}\right|$ is likely to be large for all p, and that the extra votes $|\sum_t \xi_{pt} X_t|$ are likely to all be small. This argument essentially follows the structure of the proofs of correctness for the shared coins in [12] and subsequent work [4, 6, 7]. The new part is a trick for simultaneously bounding the number of extra votes observed by each process p after time κ_p, based on the effect of the random sampling of the done bits.

3.4 Bound on Common Votes

To simplify the argument, we concentrate on the case where all processes see a positive total vote; the negative case is symmetric. First we consider the effect of the pre-threshold votes:

Lemma 3. *For sufficiently large n,* $\Pr[S_T \geq 8n] \geq 1/8$.

Proof. Immediate from the normal approximation to the binomial distribution: $8n = \sqrt{T}$ is one standard deviation.

For the remainder of the proof we consider only events that occur after the T-th vote is generated. The bounds we obtain will thus hold independently of the value of S_T.

First, we use Kolmogorov's inequality (following the approach in [7]) to bound how far S_t can drop after S_T.

Lemma 4

$$\Pr\left[\min_{t \geq T} (S_t - S_T) \leq -4n\right] \leq \frac{1}{8}.$$

Proof. From Lemma 2, there are at most $n^2 + n$ votes after T, giving a total variance of at most $n^2 + n$. Thus,

$$\begin{aligned}
\Pr\left[\min_{t \geq T} (S_t - S_T) \leq -4n\right] &\leq \Pr\left[\max_{t \geq T} |S_t - S_T| \geq 4n\right] \\
&\leq \frac{n^2+n}{(4n)^2} \\
&= \frac{1}{16} + \frac{1}{16n} \\
&\leq \frac{1}{8},
\end{aligned}$$

where the second inequality follows from Kolmogorov's inequality and the last inequality holds for $n \geq 1$.

In particular, we have that, with probability at least $7/8$, $S_{\kappa_p} \geq S_T - 4n$ for all p.

3.5 Bound on Extra Votes

We now consider the votes generated after a process detects termination but before it finishes its final collect in Line [1]; i.e., those votes X_t for which $\xi_{pt} = 1$. Notice that for each p and t, the value of ξ_{pt} is determined before X_t is generated; formally, ξ_{pt} is measurable $\mathcal{F}_{t-1}$, where $\mathcal{F}_{t-1}$ records all events prior to the generation of X_t. It follows that $\mathrm{E}[\xi_{pt} X_t | \mathcal{F}_{t-1}] = \xi_{pt}\,\mathrm{E}[X_t|\mathcal{F}_{t-1}] = 0$, since $\mathrm{E}[X_t|\mathcal{F}_{t-1}] = 0$. This implies that extra votes observed by p form a martingale difference sequence and their sum can be bounded using Azuma's inequality [10]. If $n_p = \sum \xi_{pt}$, then $\Pr\left[\sum \xi_{pt} X_t \leq -2n\right] \leq \exp(-4n^2/n_p^2)$, and the probability that any process p observes $\sum \xi_{pt} X_t \leq -3n$ is bounded by $\sum_p \exp(-9n^2/n_p^2)$ by the union bound. The main trick is to show that, with constant probability, most n_p values are small enough that this sum is a small constant.

The basic idea behind this trick is that if a particular vote generated by some process q might be an extra vote for many processes, then these processes must all have written their done bits, making it more likely that q will read **true** on its next probe of the done array and finish. In particular, this means that the k-th process to write its done bit will observe at most n/k extra votes on average from q, and n^2/k extra votes on average from all processes. By itself, this would give a probability-(1/2) bound of $2n^2/k$ on the number of extra votes observed by p_k using Markov's inequality. But in fact we can show the stronger result that this bound holds simultaneously for all p_k with probability 1/2, by bounding the sum of the number of termination bits set when each vote is generated and arguing that each of p_k's extra votes contributes at least k to this sum. The result is the following:

Lemma 5. *Let p_k be the k-th process to write its* done *bit. With probability at least* $1/2$*, it holds simultaneously for all k that $n_{p_k} = \sum_t \xi_{p_k t} \leq 2n^2/k$.*

Proof. For each vote X_t, let its **contribution** c_t be $\sum_k \xi_{p_k t}$. Consider the sequence of votes $X_{t_1}, X_{t_2}, \ldots$ generated by some single process q. If one of these votes X_{t_i} has contribution w_{t_i}, then q's next probe of the done array will find **true** with probability at least k/n; in other words, with probability at least k/n, a contribution k vote is the last vote q generates. Letting C be the expected total contribution of q's votes, we get the recurrence

$$C \leq k + (1 - k/n)C,$$

which has the solution

$$C \leq n,$$

no matter how the adversary chooses k. It follows that the expected total contribution of all votes cast by q is at most n, and thus that the expected total contribution $\sum_t c_t$ of all votes cast by all processes is at most n^2. By Markov's inequality, we have

$$\Pr\left[\sum_t c_t \leq 2n^2\right] \geq 1/2.$$

Now observe that process p_k only observes extra votes of contribution k or greater, since all other votes were generated before p_k wrote its done bit. So $\xi_{p_k t} = 1$ implies $c_t \geq k$, and thus

$$\begin{aligned}\sum_t \xi_{p_k t} &= \frac{1}{k} \sum_t \xi_{p_k t} k \\ &\leq \frac{1}{k} \sum_t \xi_{p_k t} c_t \\ &\leq \frac{1}{k} \sum_t c_t.\end{aligned}$$

This holds for all k. So in the event that $\sum_t c_t \leq 2n^2$, which occurs with probability at least $1/2$, we have $\sum_t \xi_{p_k t} \leq 2n^2/k$ for all k.

We now bound the extra votes for each process. To avoid dependencies, we define a truncated version of the extra votes for each process p_k, capped at $\lfloor 2n^2/k \rfloor$ votes; when the bound in Lemma 5 holds, this will be equal to the actual extra votes. Let $\xi'_{p_k t} = \xi_{p_k t}$ if $\left(\sum_{s<t} \xi_{p_k s}\right) < \lfloor 2n^2/k \rfloor$ and 0 otherwise. Then $\xi'_{p_k t}$ is predictable and the sequence $\left\{\sum_{i \leq t} \xi'_{p_k i} X_i\right\}$ is a martingale. If we consider just the sequence of values $X_{i_1}, X_{i_2}, \ldots$ for which $\xi'_{p_k i} = 1$, we have a bounded martingale difference sequence of length at most $2n^2/k$. It follows from Azuma's inequality that

$$\begin{aligned}\Pr\left[\sum_t \xi'_{p_k t} X_t \leq -3n\right] &\leq \exp\left(-\frac{(3n)^2}{2(2n^2/k)}\right) \\ &= e^{-(9/4)k}. \qquad (1)\end{aligned}$$

Summing this quantity for all k gives

Lemma 6. $\Pr\left[\exists k \sum_t \xi'_{p_k t} X_t \leq -3n\right] < 1/8$.

Proof. Using the union bound and (1):

$$\begin{aligned}\Pr\left[\exists k \sum_t \xi'_{p_k t} X_t \leq -3n\right] &\leq \sum_{k=1}^{n} \Pr\left[\sum_t \xi'_{p_k t} X_t \leq -3n\right] \\ &\leq \sum_{k=1}^{n} e^{-(9/4)k} \\ &= \sum_{k=1}^{n} \left(e^{-9/4}\right)^k \\ &\leq \frac{e^{-9/4}}{1 - e^{-9/4}} \\ &\approx 0.1178\ldots \\ &< 1/8.\end{aligned}$$

3.6 Full Result

We can now state the full result:

Theorem 1. *Algorithm 1 implements a shared coin with agreement parameter* $1/32$.

Proof. From Lemma 3, the probability that $S_T \geq 8n$ is at least $1/8$. Suppose that this event occurs; we now consider the probability that subsequent events involving $X_{T+1} \dots$ cause some process to compute a non-positive total vote.

Recall that each process p observes a total vote $S_{\kappa_p} + \sum_t \xi_{pt} X_t$, and that the actual vote computed by the process may be as low as $S_{\kappa_p} + \sum_t \xi_{pt} X_t - (n-1)$ (the $n-1$ is from the unwritten votes described in Lemma 1). Lemma 4 gives a probability of at most $1/8$ that any S_{κ_p} is less than $S_T - 4n$. Lemma 5 gives a probability of at most $1/2$ that $\sum_t \xi_{pt} \neq \sum_t \xi'_{pt}$ for any p. Finally, Lemma 6 gives a probability of at most $1/8$ that $\sum_t \xi'_{pt} \leq -3n$ for any p. These probabilities sum to $3/4$; so there is a probability of at least $1/4$ that none of these bad events occur, in which case the total vote computed by p is at least $S_T - 4n - 3n - (n-1) = S_T - 8n + 1$.

When $S_T \geq 8n$, this quantity is at least 1, and thus all processes return $+1$ with probability at least $(1/8)(1/4) = 1/32$. That all processes return -1 with at least the same probability follows from symmetry of the algorithm.

Corollary 1. *There is a wait-free randomized shared-memory consensus protocol using only-single writer registers that executes* $O(n^2)$ *expected total operations against an adaptive adversary.*

4 Conclusions

We have shown that the expected total work needed for randomized shared-memory consensus with an adaptive adversary is $O(n^2)$ using only single-writer registers, matching the upper and lower bounds for multi-writer registers of Attiya and Censor [7]. This leaves the question of what happens with individual work. Here the best known lower bound is $\Omega(n)$, which matches the Attiya-Censor lower bound and which is immediate from the need for a process in a solo execution to read at least one register belonging to each other process before deciding. The best known upper bound is given by an $O(n \log^2 n)$ algorithm of Aspnes and Waarts [6] that modifies the Bracha-Rachman protocol by having processes generate votes of increasing weight. We suspect that the gossip-based termination detector used here might be able to reduce this upper bound to $O(n \log n)$, but that closing the gap completely will likely require new techniques.

Acknowledgments. I would like to thank Hagit Attiya and Keren Censor for suggesting the possibility of faster single-writer consensus and for several useful discussions.

References

1. Abrahamson, K.: On achieving consensus using a shared memory. In: Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 291–302 (1988)
2. Alon, N., Spencer, J.H.: The Probabilistic Method. John Wiley & Sons, Chichester (1992)
3. Aspnes, J.: Time- and space-efficient randomized consensus. Journal of Algorithms 14(3), 414–431 (1993)
4. Aspnes, J., Censor, K.: Approximate shared-memory counting despite a strong adversary. In: SODA 2009: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms, pp. 441–450. Society for Industrial and Applied Mathematics, Philadelphia (2009)
5. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms 11(3), 441–461 (1990)
6. Aspnes, J., Waarts, O.: Randomized consensus in expected $O(N \log^2 N)$ operations per processor. SIAM Journal on Computing 25(5), 1024–1044 (1996)
7. Attiya, H., Censor, K.: Tight bounds for asynchronous randomized consensus. Journal of the ACM 55(5), 20 (2008)
8. Attiya, H., Dolev, D., Shavit, N.: Bounded polynomial randomized consensus. In: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14–16, pp. 281–293 (1989)
9. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. John Wiley & Sons, Chichester (2004)
10. Azuma, K.: Weighted sums of certain dependent random variables. Tôhoku Mathematical Journal 19(3), 357–367 (1967)
11. Bracha, G., Rachman, O.: Approximated counters and randomized consensus. Technical Report 662, Technion (1990)
12. Bracha, G., Rachman, O.: Randomized consensus in expected $O(n^2 \log n)$ operations. In: Toueg, S., Spirakis, P.G., Kirousis, L.M. (eds.) WDAG 1991. LNCS, vol. 579, pp. 143–150. Springer, Heidelberg (1992)
13. Chor, B., Israeli, A., Li, M.: Wait-free consensus using asynchronous hardware. SIAM J. Comput. 23(4), 701–712 (1994)
14. Dwork, C., Herlihy, M., Plotkin, S., Waarts, O.: Time-lapse snapshots. SIAM Journal on Computing 28(5), 1848–1874 (1999)
15. Grimmett, G.R., Stirzaker, D.R.: Probability and Random Processes. Oxford University Press, Oxford (2001)
16. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
17. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995)
18. Saks, M., Shavit, N., Woll, H.: Optimal time randomized consensus—making resilient algorithms fast in practice. In: Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 351–362 (1991)

Structured Derivation of Semi-Synchronous Algorithms

Hagit Attiya[1,⋆], Fatemeh Borran[2], Martin Hutle[3,⋆⋆], Zarko Milosevic[2], and André Schiper[2]

[1] Technion, Israel
hagit@cs.technion.ac.il
[2] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{fatemeh.borran,zarko.milosevic,andre.schiper}@epfl.ch
[3] Fraunhofer SIT, Germany
martin.hutle@epfl.ch

Abstract. The semi-synchronous model is an important middle ground between the synchronous and the asynchronous models of distributed computing. In this model, processes can detect (timeout) when other processes fail. However, since detection is done by timing out, it incurs a cost much higher than the typical delay of messages.

The paper presents a new communication primitive, *Timely Announced Broadcast* (*TAB*), and uses it in algorithms for consensus and set consensus in the semi-synchronous model. Separate implementations of TAB, withstanding different types of failures, allow to derive algorithms for consensus and set consensus under crash and omission failures.

The time bounds obtained by our algorithms asymptotically match, or improve, the previously known bounds.

Keywords: semi-synchronous systems, timely announced broadcast, terminating reliable broadcast, set consensus.

1 Introduction

Most research in distributed computing considers two models, *synchronous* and *asynchronous*. In the synchronous model, processes take steps in rounds, and messages sent in one round are received by the next round; in the asynchronous model, processes take steps at arbitrary times, and there is no upper bound on message delay. Practical systems, however, are neither as predictable as the synchronous model nor as unpredictable as the asynchronous one. An important middle ground is the *semi-synchronous* model [4] in which there are bounds on the time processes take steps, or message are delivered, but they are only approximately known.

In the context of fault-tolerant distributed algorithms [3], it is assumed that consecutive steps by a correct process require time at least c_1 and at most c_2,

⋆ Part of this work done while visiting EPFL.
⋆⋆ Part of this work done while at EPFL.

while messages are delivered within at most time d after they are sent. When failures are benign, i.e., stopping to take steps or omitting to send/receive messages, they can be detected by timing out the faulty process, according to these time bounds, since they stop sending messages. However, to avoid suspecting the innocents, timing out a process must take a relatively long time, which we denote $TO(d)$, and assume $TO(d) > d$, typically by a large margin (this is discussed further in Section 2).

A classical problem in fault-tolerant distributed computing is the *k-set consensus* problem [7], in which processes are required to output at most k *different* values; if a correct process outputs v then v must be the input of some process. The *consensus* problem is a special case where $k = 1$. In the synchronous model, exactly $\lfloor f/k \rfloor + 1$ rounds are required for solving k-set consensus in the presence of f crash failures, for any $f < n$ [7,8]. This extends the bound for the well-known consensus problem (where $k = 1$), which can be solved in exactly $f + 1$ rounds (e.g., [10]).

These round-based algorithms can easily be *simulated* in the semi-synchronous model, by waiting to timeout all processes that did not send a message in a round, before proceeding to the next round. In this manner, executing an r-round synchronous algorithm takes roughly $r\,TO(d)$ time, namely, the timeout cost is paid for each round.

Perhaps surprisingly, this cost is not inherent, and it has been shown by Attiya, Dwork, Lynch and Stockmeyer [3] that the consensus problem can be solved in time $2fd + TO(d)$, that is, the timeout cost is paid only once (this algorithm is called ADLS). They also show that the timeout cost must be paid at least once, by proving a time lower bound of $(f-1)d + TO(d)$. In follow-up work, Michailidis [17] presented an algorithm solving set consensus within time $\lfloor \frac{f}{k} \rfloor d + TO(2d)$.

Despite being simple, the structure of the ADLS algorithm is quite different from other consensus algorithms, and the way it works is considered "a mystery" (cf. [14]). This might be the reason there has been very little work in the semi-synchronous model.

Our contribution: In this paper, we present a new communication primitive, called *timely announced broadcast* (TAB), which simplifies the design of semi-synchronous algorithms for consensus and set consensus. TAB has simple implementations in different failure models, and we present two efficient ones for crash and omission failures. Combining these TAB implementations with simple consensus and set consensus algorithms lead to structured algorithms that match, or even improve, the best known time bounds for the semi-synchronous model.

Specifically, TAB provides three primitives to broadcast, announce and deliver a message. Informally, after a process q broadcasts m, all other processes first announce m and only then deliver m. In addition to common properties of broadcast primitives (like integrity and validity), it is guaranteed that if a correct process delivers m from q, then every correct process announces m from q. In this respect, TAB is similar to known primitives like *adopt-commit* (e.g., [13]) or *graded consensus* (e.g., [12]). Unlike these other primitives, however, TAB also provides timing guarantees, as it bounds the time duration between the broadcast, announcement, and delivery of the same message.

We present implementations of TAB in the presence of crash and omission failures. For crash failures, the time for both announcement and delivery is bounded by d, while for omission failures, the time for announcement is bounded by d and the time for delivery is bounded by $2d$. (The last algorithm assumes that $n > 2t$, where t is the maximum number of failures that can occur in an execution; we reserve f for the number of failures in a specific execution.)

We then show how TAB can be used in a simple flooding-style algorithm for *terminating reliable broadcast*, leading to a simple consensus algorithm with the same time bounds. A somewhat more elaborate, but still intuitive when considered in a synchronous setting, algorithm is needed for solving *set consensus* using TAB. Employing the TAB implementation for crash failures, we get a time bound of $fd + TO(2d)$ for consensus and a time bound of $\lfloor f/k \rfloor d + TO(2d)$ for set consensus. Employing the TAB implementation for omission failures, we get a time bound of $2fd + TO(4d)$ for consensus and a time bound of $2\lfloor f/k \rfloor d + TO(4d)$ for set consensus.

Prior bounds: Table 1 summarizes the time bounds of our algorithms and previous results.

A couple of papers extended [3] to omission failures. Ponzio [18] showed that when $n > 2t$, a simulation of crash failures on top of omission failures can be applied to the algorithm of [3] to derive an algorithm for omission failures requiring $4(f + 1)d + TO(d)$ time. Berman and Bharali [6] present an improved consensus algorithm for omission failures, requiring $3(f + 1)d + TO(d)$ time, when $n > 2t$. Both papers [18,6] also present more complicated bounds for the case $n \leq 2t$.

As for lower bounds, Herlihy, Rajsbaum and Tuttle [16] prove that in the semi-synchronous model, any k-set consensus algorithm for n processes and $n-1$ crash failures, requires time $\lfloor \frac{n-1}{k} \rfloor d + TO(d)$. Herlihy and Rajsbaum [15] extend this result to hold also with adversaries that fail processes in a coordinated manner. For the threshold adversary considered in our paper, where process failures

Table 1. Comparison of our results with prior results (algorithms for omission failures assume $n > 2t$)

Problem		crash failures	omission failures
Consensus	Attiya et al. [3]	$2fd + TO(d)$	
	Michailidis [17]	$fd + TO(2d)$	
	Ponzio [18]		$4(f + 1)d + TO(d)$
	Berman and Bharali [6]		$3(f + 1)d + TO(d)$
	this paper	$fd + TO(2d)$	$2fd + TO(4d)$
k-set consensus	Michailidis [17]	$\lfloor f/k \rfloor d + TO(2d)$	
	this paper	$\lfloor f/k \rfloor d + TO(2d)$	$2\lfloor f/k \rfloor d + TO(4d)$

are independent, they show a bound of $\lfloor \frac{f-1}{k} \rfloor d + TO(d)$, extending the previous result. These lower bounds show that our upper bounds are asymptotically optimal.

2 Preliminaries

Model of Computation: We use the model defined in [3], in which there are n processes $p_1, \ldots, p_n$, and their respective message buffers, buff_1, …, buff_n. Each process p_i is modeled as a (possibly infinite) state machine with a local state set Q_i, including a distinguished *initial state*.

A *configuration* is a vector $s = ((q_1, b_1) \ldots, (q_n, b_n))$ where $\mathit{state}_i(s) = q_i$ is the local state of p_i and $\mathit{buff}_i(s) = b_i$ is the content of p_i's buffer. In the *initial configuration* all processes are in their initial states and all buffers are empty.

Processes communicate by sending *messages*. We assume that messages sent from p_i to p_j contain a sequence number and that the sender's id is part of every message. The action $\mathit{send}(p_j, m)$ represents the sending of message m to process p_j.

Each process p_i follows a deterministic algorithm that governs its state transitions and the messages it sends. The possible events are either *computation events* of the form $\mathit{comp}(p_i, S)$, where p_i is a process and S is a set of send actions, or *delivery events* of the form $\mathit{deliv}(p_i, m)$, where p_i is a process and m is a message.

An *execution* is an infinite sequence of alternating configurations and events $\alpha = C_0, \pi_1, C_1, \ldots, \pi_r, C_r, \ldots$, satisfying the following conditions:

1. C_0 is the initial configuration.
2. If π_r is an event of process p_i, then $\mathit{state}_j(C_{r-1}) = \mathit{state}_j(C_r)$ and $\mathit{buff}_j(C_{r-1}) = \mathit{buff}_j(C_r)$ for every $j \neq i$. That is, states and buffers of processes other than p_i do not change.
3. If $\pi_r = \mathit{comp}(p_i, S)$, then $\mathit{state}_i(C_r)$ and S are obtained by applying γ_i to $\mathit{state}_i(C_{r-1})$ and $\mathit{buff}_i(C_{r-1})$; furthermore, $\mathit{buff}_i(C_r) = \emptyset$. That is, p_i, based on its local state and the contents of its buffer, performs the send actions in S, clears its buffer and possibly changes its local state, all in one atomic transition.
4. If $\pi_r = \mathit{deliv}(p_i, m)$, then $\mathit{state}_i(C_r) = \mathit{state}_i(C_{r-1})$ and $\mathit{buff}_i(C_r) = \mathit{buff}_i(C_{r-1}) \cdot \{m\}$. That is, the message m is appended to p_i's buffer.
5. For every delivery event $\pi_r = \mathit{deliv}(p_i, m)$ there is exactly one computation event $\pi_l = \mathit{comp}(p_j, S)$ where $l < r$ and $\mathit{send}(p_i, m) \in S$. That is, each delivery is matched to a unique earlier send.

Below, 'time' is always a nonnegative real number. A *timed event* is a pair (π, t), where π is an event and t is a time. A *timed execution* is an infinite sequence of alternating configurations and timed events $\alpha = C_0, (\pi_1, t_1), C_1, \ldots, (\pi_r, t_r), C_r, \ldots$, where $C_0, \pi_1, C_1, \ldots, \pi_r, C_r, \ldots$ is an execution and the times are nondecreasing and unbounded.

Types of Failures: Fix real numbers c_1, c_2, and d, $0 < c_1 \leq c_2 < \infty$ and $0 < d < \infty$. A process p_i is *correct* in a timed execution α if the following conditions hold:

1. There is a computation event $comp(p_i, S)$ at time 0.
2. If the lth and rth timed events, $l < r$, are both computation events of p_i with no intervening computation events of p_i, then $c_1 \leq t_r - t_l \leq c_2$.
3. If a message m is sent by p_i to p_j at the lth timed event then there exists $r > l$ such that the rth timed event is the matching delivery $deliv(p_i, m)$, and $t_r - t_l \leq d$.

If a process is not correct, we say it is *faulty*, and denote by t the largest number of faulty processes that the protocol has to tolerate.

We model failure types by restricting the behavior of a faulty process. We only consider *benign* failures, where faulty processes follow the algorithm. With *crash failures* a faulty process may stop taking steps (or not start at all), but its messages are delivered on time. Specifically, every message that is sent at time T is received the latest at time $T + d$, if the receiver process is correct.

Omission failures are slightly more severe than crash failures, and although they ensure the delivery times of messages, messages sent by a faulty process or to faulty process may not be delivered at all. Specifically, every message that is received at time T is sent not before time $T - d$. Every message that is sent by a correct process is received if the receiver is correct.

Following the literature on *early-stopping* consensus and set consensus, we denote by f the number of processes that fail in a specific execution of the algorithm, assuming that the execution is clear from the context. We reserve t to denote the maximum number of failures that is possible in all executions.

A Timeout Task: Let $TO(T)$ be the worst-case time to detect that time T has elapsed. In order to ensure that time T has elapsed, a process must count $\frac{T}{c_1}$ of its own steps,[1] since it might be running "fast" (i.e., time c_1 between steps). But if the process is actually running "slow" (i.e., time c_2 between steps), the actual waiting time is $\frac{c_2 T}{c_1}$. That is, $TO(T) = CT$, with $C = \frac{c_2}{c_1}$.

In order to detect the failure of process p, processes must ensure that no messages from p are in transit. Therefore, the worst-case elapsed time between the failure of p and the time when all correct processes determine that p has failed is roughly Cd.

3 Timely Announced Broadcast

We introduce a new communication primitive, *Timely Announced Broadcast (TAB)*, and show how it can be used to solve the consensus and set consensus problems. TAB is defined in terms of three primitives, *ta-broadcast*(m),

[1] We ignore rounding issues and assume that c_1 always divides T and c_2.

Algorithm 1. TAB with crash failures; code for process p

```
upon ta-broadcast($m$) do
    send $\langle$ANNOUNCE$, m, p\rangle$ to all
    send $\langle$MSG$, m, p\rangle$ to all

upon received $\langle$ANNOUNCE$, m, q\rangle$ do
    if not announced $m$ from $q$ yet then
        announce($m, q$)

upon received $\langle$MSG$, m, q\rangle$ do
    if not announced $m$ from $q$ yet then
        announce($m, q$)
    ta-deliver($m, q$)
```

announce(m, q), and *ta-deliver*(m, q). Informally, after a correct process q invokes *ta-broadcast*(m), all other processes first *announce*(m, q) and only then *ta-deliver*(m, q).

The announce message indicates to a process to wait for a forthcoming message, causing it to extend its timeout and wait enough time to deliver the expected message (as demonstrated in Section 4).

In addition to common properties of broadcast primitives (like integrity and validity), it is guaranteed that if a correct process ta-delivers m from p, then every correct process announces m from p. TAB also provides timing guarantees, as it bounds the time duration between the broadcast, announcement, and delivery of the same message. In our implementations of TAB, these bounds are in $O(d)$, with the constant being small, i.e., 1 or 2.

Definition 1 (TAB). *An algorithm solves* timely announced broadcast in the presence of benign failures*, with two parameters $d_1 \geq d_2 > 0$, if the following properties hold:*

Integrity. *If a process ta-delivers a message m from p, then m was ta-broadcast by p.*

Validity. *If a correct process p ta-broadcasts a message m at time T, then all correct processes eventually announce m from p and ta-deliver m from p the latest at time $T + d_1$.*

Announcement. *For any message m, if any process ta-delivers m from p at time T, then every correct process announces m from p the latest at time $T + d_2$.*

Implementing TAB in the presence of crash failures: Algorithm 1 implements TAB, with crash failures and assuming $n > t$.

It is easy to verify that the algorithm satisfies the properties of Definition 1, with $d_1 = d_2 = d$.

Implementing TAB in the presence of omission failures: Algorithm 2 implements TAB, with omission failures and assuming $n > 2t$. It is simple to show that the

Algorithm 2. TAB with omission failures and $n > 2t$; code for process p

```
1: upon ta-broadcast($m$) do
2:     send $\langle \textsc{Msg}, m \rangle$ to all

3: upon received $\langle \textsc{Msg}, m \rangle$ from $q$ do
4:     send $\langle \textsc{Ack}, m, q \rangle$ to all

5: upon received $\langle \textsc{Ack}, m, q \rangle$ the first time do
6:     announce($m, q$)

7: upon received $t + 1$ $\langle \textsc{Ack}, m, q \rangle$ do
8:     ta-deliver($m, q$)
```

algorithm satisfies the properties of Definition 1, with $d_1 = 2d$ and $d_2 = d$. Inspecting the code verifies that the integrity property holds.

Lemma 1 (Integrity). *If a process ta-delivers m from p, then m was ta-broadcast by p.*

Lemma 2 (Validity). *If a correct process p ta-broadcasts a message m at time T, then all correct processes eventually announce m from p and ta-deliver m from p the latest at time $T + d_1$, where $d_1 = 2d$.*

Proof. Since p is correct and ta-broadcasts m at time T, it sends $\langle \textsc{Msg}, m \rangle$ to all by time T, and by time $T + d$, all correct processes send $\langle \textsc{Ack}, m, p \rangle$ to all.[2] Since $n > 2t$, this means that by time $T + 2d$, every process receives at least $n - t \geq t + 1$ $\langle \textsc{Ack}, m, p \rangle$ messages and therefore announces and ta-delivers m from p. □

Lemma 3 (Announcement). *For any message m, if any process ta-delivers m from p at time T, then every correct process announces m from p the latest at time $T + d_2$, where $d_2 = d$.*

Proof. If a process delivers m from p at time T, then it received $t+1$ $\langle \textsc{Ack}, m, p \rangle$ messages by time T, at least one of them was sent by a correct process q by time T. Then by time $T + d$, every correct process receives an $\langle \textsc{Ack}, m, p \rangle$ message from q and announces m from p. □

4 Terminating Reliable Broadcast from TAB

The *Terminating Reliable Broadcast (TRB)* problem is defined in terms of two primitives, broadcast and deliver, and has a dedicated sending process s. The sender process s is the only process that invokes *broadcast*. When failures are benign, we have the following requirements.

[2] To simplify the statements of the results and the proofs, we assume $c_2 \ll d$ and approximate $d + c_2$ with d.

Definition 2 (TRB). *An algorithm solves* terminating reliable broadcast in the presence of benign failures *if the following properties hold:*

Integrity. *A process delivers at most one message, and if a process delivers a message $m \neq \perp$, then m was broadcast by s.*
Validity. *If the sender s is correct and broadcasts a message m, then s eventually delivers m.*
Agreement. *If a correct process delivers a message m, then every correct process delivers m.*
Termination. *Every correct process eventually delivers some message.*

Consensus from TRB: Non-uniform consensus can be easily implemented from TRB with a simulation, where n instances of the TRB algorithm are executed in parallel, in each instance i process i is the sender and uses its initial value for that, and all processes apply a deterministic function on the resulting vector. The response time of this consensus algorithm equals the response time of the TRB algorithm.

TRB from TAB: It is easy to solve TRB in a synchronous system, using a familiar, simple *flooding* mechanism. In the simplest form of this protocol (cf. [5, Algorithm 15]), for a synchronous system with crash failures, the sender sends a message to all processes; each process that gets this message, echoes it by re-sending it to all processes, and returns the value. If no value was returned after $f+1$ rounds, the process returns $\perp$.

This algorithm can be deployed in a semi-synchronous system by timing out the processes that failed at the beginning of the round before correct processes advance to the next round. This algorithm however, is susceptible to timing delays, since timing out a process takes $TO(d)$ and hence timing out r rounds may take $r\,TO(d)$ time, yielding a $(f+1)Cd$-time algorithm for consensus.

To overcome this problem, instead of sending and receiving messages directly, processes use TAB to send and announce-deliver messages. The TAB protocol allows processes to warn other processes that they are about to deliver a message, thus alerting them to wait for their copy of this message.

The pseudocode is given as Algorithm 3. For a process p, Z_p holds the set of processes who have sent an announcement and p must wait for their message. For any process $q \in Z_p$, if process p does not receive a message from q within a specific time, namely $2d_1$, it suspects q to be faulty and stops waiting for a message from q by removing q from Z_p (line 11). When $Z_p = \perp$, all processes who have only sent an announcement are crashed, so it is safe to deliver $\perp$ (lines 12-13). We show the algorithm solves TRB.

Lemma 4 (Integrity). *A process delivers at most one message, and if a process delivers a message $m \neq \perp$, then m was broadcast by s.*

Proof. The first part of the lemma follows from the code (a process stops after delivering a message). Assume that a process p delivers $m \neq \perp$. Since a non-$\perp$ message is delivered only after a *ta-deliver* event at line 9, this means that

Algorithm 3. TRB from TAB; code for process p (s is the sender process)

```
1:  initially
2:     $Z_p \leftarrow \{s\}$

3:  upon broadcast $v$ do                                        /* called only by $s$ */
4:     ta-broadcast($v$)

5:  upon announce($m, q$) do          /* a message from $q$ is forthcoming, wait for it */
6:     $Z_p \leftarrow Z_p \cup \{q\}$

7:  upon ta-deliver($v, q$) the first time
     do                          /* a message from $q$ ta-delivered, echo it and deliver */
8:     ta-broadcast($v$)
9:     deliver $v$

10: upon no ta-deliver($v, q$) for $2d_1$ time since the last announce($v, q$) or time 0
     do                                                               /* suspect $q$ */
11:    $Z_p \leftarrow Z_p \setminus \{q\}$                /* stop waiting for all messages from $q$ */
12:    if $Z_p = \emptyset$ then
13:       deliver $\perp$
```

p ta-deliver (m, q). By the Integrity property of TAB, m was ta-broadcast by process q. If $q = s$, this finishes the proof. Otherwise, q ta-broadcast value it ta-deliver from some process. Either q ta-deliver message from s or after a chain of processes, where by the Integrity property of TAB and the fact that no process executes line 8, at least one of processes in the chain will ta-deliver message from process s. Therefore, if p deliver a non-$\perp$ message, then it must be m. □

Lemma 5 (Validity). *If the sender s is correct and broadcasts a message m, then s eventually delivers m.*

Proof. From the Validity property of TAB, s ta-delivers m the latest at time d_1. Since at this time s cannot execute the upon rule of Line 10, s delivers m by Line 9. □

Lemma 6 (Agreement). *If a correct process delivers a message m, then every correct process delivers m.*

Proof. By contradiction. Because of Lemma 4, w.l.o.g. assume that at time T_p, a correct process p delivers v, the value broadcast by s, and at time T_q, a correct process $q \neq p$ delivers $\perp$.

If process q delivers $\perp$ at time T_q, it did not ta-deliver nor receive an announcement for $2d_1$ time (a ta-delivery would have led to deliver v and an announcement would have delayed the delivery of $\perp$). Thus, process p cannot deliver before time $T_q - d_1$: before delivering it would have ta-broadcast its message to all, and thus, since p is correct, q would have received this message. Therefore, $T_p > T_q - d_1$. Because p delivered v, it ta-delivered a message; in more detail, there is a chain of deliveries from the source to p. Each of these ta-broadcast events is at most

d_1 apart from each other. Since $T_q \geq d_1$, and the source ta-broadcasts at time 0, there is one process in this chain that ta-broadcasts after $T_q - 2d_1$ but before $T_q - d_1$. Because of the Announcement property, an announcement is received by q between $T_q - 2d_1$ and $T_q - d_1 + d_2 \leq T_q$. This contradicts the fact that q neither ta-delivers nor receives an announcement between $T_q - 2d_1$ and T_q. □

The upon rule of Line 10 ensures the next lemma.

Lemma 7 (Termination). *Every correct process eventually delivers some message.*

The next lemma shows the timing property of the TRB algorithm.

Lemma 8. *In a run with an actual TAB transmission time d_1 and f faulty processes that obey the timing requirements, a correct process delivers a value by time $fd_1 + TO(2d_1)$.*

Proof. By Lemma 7, a correct process p eventually delivers a value for the sender s. If p delivers $m \neq \perp$ in Line 9, then it can be easily shown (cf. [5, Algorithm 15]) that it has received m along a chain of r re-broadcasts by different processes. It follows that $r \leq f + 1$, since once m reaches a correct process it is sent to all processes. Thus, it is delivered by time $(f + 1)d_1 = fd_1 + d_1 \leq fd_1 + TO(d_1)$ (since $TO(d) \geq d$).

Consider now the case p delivers $\perp$ in Line 13; we argue that each process q added to Z_p is removed by time $fd_1 + TO(2d_1)$, and the upon rule of Line 10 ensures $\perp$ is delivered.

A process q is added to Z_p in the upon rule of Line 5. Since no process announces the same process twice (by Line 7), an announcement is forwarded through a chain of r different processes $p_{i_1}, \ldots, p_{i_r}$.

If none of these processes is correct, then $r \leq f$, and hence q is added in the upon rule of Line 5 before time fd_1, and it is timed out (in the upon rule of Line 10) before time $fd_1 + TO(2d_1)$, implying the claim.

So, let $p_{r'}$ be the first correct process among $p_{i_1}, \ldots, p_{i_r}$; clearly, $r' \leq f + 1$, and hence p receive the ta-broadcasts from $p_{r'}$ before time $r'd_1 \leq (f + 1)d_1 \leq fd_1 + TO(2d_1)$ (since $TO(d) \geq d$). □

By substituting the appropriate TAB implementations, we get:

Corollary 1. *There is a TRB algorithm, and hence consensus algorithm, which withstands crash failures and terminates within time $fd + TO(2d)$.*

Corollary 2. *There is a TRB algorithm, and hence consensus algorithm, which withstands omission failures and terminates within $2fd + TO(4d)$, assuming that $n > 2t$.*

5 Set Consensus from TAB

Definition 3 (Set consensus). *An algorithm solves k-*set consensus in the presence of benign failures *if each process starts with an input value, and decides on a value, such that the following properties hold:*

Integrity. *If a process decides v, then v is the input of some process.*
Agreement. *The correct processes decide on at most k different values.*
Termination. *Every correct process eventually decides.*

The pseudocode for set consensus appears in Algorithm 4. Recall that all processes start at time 0. Each process p keeps an array of known initial values; $know_p[q]$ is the initial value of process q learned by p, and is initially $\perp$. During the algorithm, process p *learns* the initial values of other processes. Unlike TRB that has a unique source, in set consensus all processes are sources. Therefore, process p keeps a set Z_p for each process q: $Z_p[q]$ denotes the set of processes who sent an announcement for q, that is, $Z_p[q]$ is the set of processes who have

Algorithm 4. Set Consensus from TAB; code for process p

```
initially
    $\forall q \in \Pi : know_p[q] \leftarrow \perp$
                    /* $know_p[q]$ contains the initial value of process $q$ learned by $p$ */
    $\forall q \in \Pi : Z_p[q] \leftarrow \{q\}$
          /* set of processes that announced the knowledge of the initial value of $q$ */

upon starting with input $v$ do
    $know_p[p] \leftarrow v$
    $ta\text{-}broadcast(know_p)$

upon $announce(kn, q)$ do
    for all $r : kn[r] \neq \perp$ do
        $Z_p[r] \leftarrow Z_p[r] \cup \{q\}$
                    /* $q$ learned $r$'s initial value but $p$ didn't, wait for this value */

upon $ta\text{-}deliver(kn, q)$ do
    for all $r \in \Pi$ do
        if $know_p[r] = \perp$ and $kn[r] \neq \perp$ then
            $know_p[r] \leftarrow kn[r]$                    /* learn initial values $q$ knows */
    if updated $know_p$ then
        $ta\text{-}broadcast(know_p)$       /* inform others about new learned initial values */
    if can-decide? then
        decide $\min(know_p)$

upon no $ta\text{-}deliver(v, q)$ for $2d_1$ time since the last $announce(v, q)$ or time 0
  do                                                                 /* suspect $q$ */
    for all $r \in \Pi$ do
        $Z_p[r] \leftarrow Z_p[r] \setminus \{q\}$              /* stop waiting for all messages from $q$ */
    if can-decide? then
        decide $\min(know_p)$

function can-decide?
    $K_p \leftarrow \{q : know_p[q] = \perp\}$ /* set of processes that $p$ doesn't know their value */
    $U_p \leftarrow \{q : \exists r$ s.t. $know_p[r] = \perp \ \wedge \ q \in Z_p[r]\}$
     /* set of processes that know values of those processes that $p$ doesn't know */
    return $(|K_p| < k)$ or $(|U_p| < k)$
```

learned the initial value of process q. As for Algorithm 3, if process p does not receive a message from q within the specified time, namely $2d_1$, it suspects q to be faulty and stops waiting for a message from q (lines 18-20).

The decision condition is also more complicated that in TRB. As in synchronous set consensus algorithms [7], process p can decide if it knows more than $n-k$ values, i.e., $|K_p| < k$, where K_p denotes the set of processes that p does not know their initial value. Additionally, process p can decide if fewer than k processes know the initial value of those processes that p does not know, i.e., the size of $U_p \triangleq \{q : \exists r \text{ s.t. } \mathit{know}_p[r] = \perp \ \wedge \ q \in Z_p[r]\}$ is strictly less than k. That is, if only $k-1$ processes know some values that p does not know, at most $k-1$ different values might be decided; therefore, p can decide on the minimum value that it knows.

The correctness proof follows arguments similar to those used to prove the correctness of Algorithm 3. A process decides only on a value from the *know* array; the values in this array are either the process's initial value or those received in a *ta-deliver* event. Hence, the Integrity property of TAB implies the next lemma.

Lemma 9 (Integrity). *If a process decides v, then v was proposed by some process.*

Lemma 10 (Agreement). *The correct processes decide on at most k different values.*

Proof. Assume, towards a contradiction, that at least $k+1$ different values, $v_1 < \ldots < v_{k+1} < \ldots$, are decided. Let p be a correct process that decides v_{k+1}. By Line 26, p decides either because $|K_p| < k$ or because $|U_p| < k$.

If $|K_p| < k$, then since p decides on the minimum value it has seen it follows that it has not seen $v_1, \ldots, v_k$, that is, k values, which is a contradiction.

Otherwise, p decides because $|U_p| < k$. Note that this happens only when the condition of Line 18 is satisfied, i.e., $2d_1$ timeout expires. $|U_p| < k$ implies that (i) at most $k-1$ processes know values that p doesn't know. This means that at most k different values are decided (including p's decision value). By the assumption, (ii) there are at least $k+1$ decisions.

From (i) and (ii), it follows that there is a decision value $x < v_{k+1}$, such that every process in U_p that knows x also knows a value smaller than x. Let q be a correct process that decides x. We consider two cases for how q has received x:

Case 1, q receives x after p decides: Since p decided v_{k+1} and $x < v_{k+1}$, p does not know x, this implies that q did not receive x from p. Therefore, q must have received x from some process in U_p, possibly through other intermediate processes. But every process in U_p that knows x also knows a value smaller than x, a contradiction.

Case 2, q receives x before p decides: q must appear in $Z_p[r]$, for some $r \in \Pi$, since q ta-broadcasts x before deciding. Since p does not know x and p has timed out all faulty processes by Line 18, p must have detected q's failure, a contradiction. □

The upon rule of Line 18 together with the properties of TAB ensure the next lemma.

Lemma 11 (Termination). *Every correct process eventually decides.*

Proof. We show that every correct process continues to take steps until it decides or crashes, i.e., the condition of Line 26 eventually becomes true.

Assume, by way of contradiction, that some correct process p continues to take steps without deciding. This means that the sizes of the sets K_p and U_p it has are greater than or equal to k (according to Line 26). Thus, there are at least k unknown values and at least k processes in $Z \triangleq \bigcup_{r \in \Pi} Z_p[r]$. Consider one of these processes, say q. Since $q \in Z$, it has ta-broadcast $know_q$. If q is a correct process, by the Validity property of TAB, p ta-delivers a message from q within d_1. Otherwise, p suspects q after $2d_1$ according to Line 18. In either case, the condition of Line 26 becomes satisfied, which contradicts the fact that p never decides. □

The final lemma shows the timing property of the set consensus algorithm.

Lemma 12. *In a run with an actual TAB transmission time d_1 and f faulty processes that obey the timing requirements, a correct process decides by time $\lfloor f/k \rfloor d_1 + TO(2d_1)$.*

Proof. (Sketch) By Lemma 11, a correct process p eventually decides. Let p decide v, where v is the initial value of some process q, i.e., $know_p[q] = v$. From Line 26, p decides because either (i) $|K_p| < k$ or (ii) $|U_p| < k$.

In case (i), it can be easily shown (cf. [7]) that p has received v along a chain of $r \leq \lfloor f/k \rfloor + 1$ re-broadcasts by different processes. Thus v is decided by time $(\lfloor f/k \rfloor + 1)d_1 \leq \lfloor f/k \rfloor d_1 + TO(d_1)$ (since $TO(d) \geq d$).

In case (ii) we show that each process $q' \in Z_p[q]$ is removed by time $\lfloor f/k \rfloor d_1 + TO(2d_1)$. Process q' receives v through a chain of r different processes.

If none of these processes is correct, then $r \leq \lfloor f/k \rfloor$. Therefore, q' is added to $Z_p[q]$ by time $rd_1 \leq \lfloor f/k \rfloor d_1$ in the upon rule of Line 7 and is removed from $Z_p[q]$, after a timeout $TO(2d_1)$, by time $rd_1 + TO(2d_1) \leq \lfloor f/k \rfloor d_1 + TO(2d_1)$ in the upon rule of Line 18, which implies the lemma.

Otherwise, p receives v from a correct process by time $r'd_1$, where $r' \leq \lfloor f/k \rfloor + 1$. Therefore, p decides v by time $(\lfloor f/k \rfloor + 1)d_1 \leq \lfloor f/k \rfloor d_1 + TO(2d_1)$ (since $TO(d) \geq d$). □

By substituting the appropriate TAB implementations, we get:

Corollary 3. *There is a k-set consensus algorithm, which withstands crash failures and terminates within time $\lfloor f/k \rfloor d + TO(2d)$.*

Corollary 4. *There is a k-set consensus algorithm, which withstands omission failures and terminates within $2\lfloor f/k \rfloor d + TO(4d)$, assuming that $n > 2t$.*

6 Summary

This paper presents a new communication primitive and uses it to derive consensus and set consensus algorithms for semi-synchronous systems, under several types of failures. The time bounds achieved by our algorithms asymptotically match or improve previously known bounds, but we consider the main contribution of our paper to be the modular structure of our algorithms, which provides insight into the behavior of efficient semi-synchronous algorithms.

The time bounds of our algorithms are the sum of two terms: one depending only on d and another depending on a timeout (which itself depends on d). Interestingly, it can be shown that the first term is even smaller in some executions. Let δ be the maximum transmission delay *of a certain execution*. Then the execution time of, e.g., our consensus algorithm for crash failures is in fact $f\delta + TO(2d)$, which is important for the case $\delta \ll d$. (The other bounds can be adjusted similarly.)

We remark that our algorithms are *early stopping*, since their time bounds depend on f, the actual number of failures in an execution, rather than on t, the maximal number of failures. Thus, overall, the execution time of these algorithms is a constant (in terms of f) plus a term that depends only on the actual properties of an execution, f and δ (and not t and d).

The most obvious open question is to tighten the time bounds, especially for omission failures.

It is also interesting to study how TAB (or some extension thereof in the style of [9]) can yield algorithms that withstand timing or Byzantine failures. Ponzio [18] showed that in the presence of Byzantine failures, consensus can be solved in $(f+1)(d+TO(d))$. Attiya and Djerassi-Shintel [2] prove lower bounds in the presence of t *timing* failures. Specifically, they showed any consensus algorithm requires $\Omega(\frac{n}{n-t}TO(d))$ time, while a k-set consensus algorithm requires $\Omega(\frac{n}{k(n-t)}TO(d))$ time. This leaves a gap for small values of t.

Taking a broader perspective, can TAB be used to derive efficient algorithms for other problems, or even as a general technique for simulating synchronous algorithms?

The *partially synchronous* model [11] considers an asynchronous system and requires algorithms to terminate only after the system experiences a long enough synchronous period. (This is also known as the *eventually synchronous* model.) It would be intriguing to investigate implementing TAB in this model, and using it to efficiently solve problems such as consensus and set consensus. A key step would be to make our algorithms work even when not all process start at the same time (non-synchronized start).

Aguilera, Le Lann and Toueg [1] show how fast failure detection can speed up consensus in a synchronous system; their results are similar to [3]. However, as explained in their paper, "specialized hardware" or "different messaging service" are required to achieve fast failure detection. This is in contrast to the ADLS model, studied in our paper, which assumes that all messages sent in this model take at most time d.

References

1. Aguilera, M.K., Lann, G.L., Toueg, S.: On the impact of fast failure detectors on real-time fault-tolerant systems. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 354–370. Springer, Heidelberg (2002)
2. Attiya, H., Djerassi-Shintel, T.: Time bounds for decision problems in the presence of timing uncertainty and failures. Journal of Parallel and Distributed Computing 61(8), 1096–1109 (2001)
3. Attiya, H., Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Bounds on the time to reach agreement in the presence of timing uncertainty. Journal of the ACM 41(1), 122–152 (1994)
4. Attiya, H., Lynch, N.A.: Time bounds for real-time process control in the presence of timing uncertainty. Information and Computation 110(1), 183–232 (1994)
5. Attiya, H., Welch, J.L.: Distributed computing: Fundamentals, simulations, and advanced topics. Wiley-Interscience, Hoboken (2004)
6. Berman, P., Bharali, A.A.: Distributed consensus in semi-synchronous systems. In: IPPS, pp. 632–635 (1992)
7. Chaudhuri, S.: More choices allow more faults: Set consensus problems in totally asynchronous systems. Information and Computation 103(1), 132–158 (1993)
8. Chaudhuri, S., Herlihy, M., Lynch, N.A., Tuttle, M.R.: Tight bounds for k-set agreement. Journal of the ACM 47(5), 912–943 (2000)
9. Coan, B.A.: A compiler that increases the fault tolerance of asynchronous protocols. IEEE Transactions on Computers 37(12), 1541–1553 (1988)
10. Dolev, D., Strong, H.R.: Authenticated algorithms for Byzantine agreement. SIAM J. Comput. 12(4), 656–666 (1983)
11. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
12. Feldman, P., Micali, S.: Optimal algorithms for Byzantine agreement. In: Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pp. 148–161 (1988)
13. Gafni, E.: Round-by-round fault detectors: unifying synchrony and asynchrony. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, PODC 1998 (1998)
14. Herlihy, M.: Public communcation. minutes 9:28–9:37, http://www.youtube.com/watch?v=s6uEsO2T2lg
15. Herlihy, M., Rajsbaum, S.: Concurrent computing and shellable complexes. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 109–123. Springer, Heidelberg (2010)
16. Herlihy, M., Rajsbaum, S., Tuttle, M.R.: Unifying synchronous and asynchronous message-passing models. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC 1998), pp. 133–142 (1998)
17. Michailidis, D.: Fast set agreement in the presence of timing uncertainty. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 249–256 (1999)
18. Ponzio, S.: Consensus in the presence of timing uncertainty: omission and byzantine failures. In: Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC 1991), pp. 125–138 (1991)

Byzantine Agreement Using Partial Authentication

Piyush Bansal[1], Prasant Gopal[2,⋆], Anuj Gupta[1], Kannan Srinathan[1], and Pranav Kumar Vasishta[1]

[1] Center for Security, Theory and Algorithmic Research, IIIT-Hyderabad, India
[2] CSAIL, MIT
piyush_bansal@research.iiit.ac.in, prasant@csail.mit.edu

Abstract. Three decades ago, Pease *et al.* introduced the problem of *Byzantine Agreement* [PSL80] where nodes need to maintain a consistent view of the world in spite of the challenge posed by Byzantine faults. Subsequently, it is well known that Byzantine agreement over a completely connected synchronous network of n nodes tolerating up to t faults is (efficiently) possible if and only if $t < n/3$. Pease *et al.* further empowered the nodes with the ability to authenticate themselves and their messages and proved that agreement in this new model (popularly known as authenticated Byzantine agreement (ABA)) is possible if and only if $t < n$. (which is a huge improvement over the bound of $t < n/3$ in the absence of authentication for the same functionality).

To understand the utility, potential and limitations of using authentication in distributed protocols for agreement, Gupta *et al.* [GGBS10] studied ABA in new light. They generalize the existing models and thus, attempt to give a unified theory of agreements over the authenticated and non-authenticated domains. In this paper we extend their results to synchronous (undirected) networks and give a complete characterization of agreement protocols.

As a corollary, we show that agreement can be *strictly* easier than all-pair point-to-point communication. It is well known that in a synchronous network over n nodes of which up to any t are corrupted by a Byzantine adversary, BA is possible only if all pair point-to-point reliable communication is possible [Dol82, DDWY93]. Thus, a folklore in the area is that maintaining *global* consistency (agreement) is at least as hard as the problem of all pair *point-to-point* communication. Equivalently, it is widely believed that protocols for BA over incomplete networks exist only if it is possible to simulate an overlay-ed complete network. Surprisingly, we show that the folklore is not always true. Thus, it seems that agreement protocols may be more fundamental to distributed computing than reliable communication.

Keywords: Byzantine agreement, reliable communication, general networks, authentication.

⋆ The author is supported by Irwin Jacobs and Klein Jacobs MIT Presidential Fellowship.

1 Introduction

Informally, the goal of *Byzantine agreement* (BA) is to maintain a consistent view of the world in spite of the challenge posed by (Byzantine) faults. The problem was first introduced by Pease *et al.* [PSL80]. They went on to show that BA(in a synchronous and non-authenticated setting) is possible if and only if 2/3 of the nodes are non-faulty. In the asynchronous setting, Fisher *et al.* [FLP85] proved the impossibility of BA tolerating even a single crash failure. Being a fundamental problem in the area of distributed algorithms, BA has been studied in wide variety of models including partially synchronous networks [DDS87], non-threshold adversaries [HM00] and mixed-adversaries [AFM99].

Owing to its high fault tolerance, an important variant on BA is the authenticated model proposed by Pease *et al.* [PSL80]. Since generation of authenticated signature for every message is costly, some works choose to consider alternatives for authentication and avoid the excess use of signatures. Specifically, Borcherding [Bor95, Bor96b] investigated the case when signatures are used in only some rounds but not all. A different approach was taken by Srikanth and Toueg [ST87] where authenticated messages are simulated by non-authenticated sub-protocols. In another line of work, Borcherding [Bor96a] considered different levels and styles of authentication and its effects on the agreement protocols. His work focuses on the properties of authentication scheme that allows us to build faster protocols for BA. Gong *et al.* [GLR95] studied the assumptions required for the authentication mechanism in protocols for BA that use signed messages. They present new protocols for BA that add authentication to oral message protocols so that additional resilience is obtained with authentication. In all, ABA has been fairly well studied by researchers.

Gupta *et al.* [GGBS10] consider the problem of Authenticated Byzantine Agreement(ABA) under a mixed adversary model. They give completeness theorems for ABA protocols over a complete network. In this work, we extend their results to arbitrarily connected (undirected) networks.

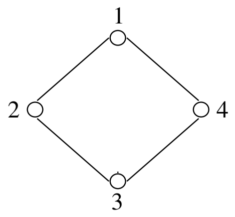


Fig. 1. Network G

1.1 Motivating Example

For starters, consider the network in Figure 1. Imagine the case when either 1 or 3 is Byzantine faulty but both of them have the ability to authenticate their messages; while 2, 4 are non-faulty but do not have the power to authenticate. We claim that in such a scenario 2 cannot reliably send a message across to 4.

In particular, consider the following executions $\mathbf{E_1}$ and $\mathbf{E_3}$: In $\mathbf{E_1}$, 2 intends to send α to 4, and during $\mathbf{E_3}$, 2 wants to send β (different from α). Now, the adversary employs the following strategy: In $\mathbf{E}_\gamma$[1], adversary corrupts γ and sends what an honest γ would have sent in $\mathbf{E}_{\overline{\gamma}}$. It can be shown that the messages received by 4 in both the executions are same and thus, 4 cannot distinguish between $\mathbf{E_1}$ and $\mathbf{E_3}$. The actual views can be proved to be same using inductive arguments. However, we do not take this up in any more detail. We, also, note that we do not use this fact elsewhere in the article. Thus, it seems that in the aforementioned scenario, nodes in G (ref. Figure 1) cannot have Byzantine agreement(BA) given the parties can't establish a reliable communication channel – which is fundamental to every distributed protocol. But, interestingly, our theorems show that nodes can agree in spite of nodes 2 and 4 not being able to establish a reliable communication link.

G turns out to be a classic example, which seems to suggest - "Perhaps, BA is more fundamental to Distributed Computing than all pair point-to-point communication". We give a simple protocol in Table 1 that solves BA over G. This is one of the many interesting examples where it seems that a BA protocol cannot exist - but our characterization gives an "efficient" way to identify if a protocol can indeed exist or not.

Organization of the Paper. In Section 2, we introduce the model. We give a summary/implications of the main theorem in Section 2.1. We, then, prove the main theorem over Sections 3, 4 and 5.

2 Model and Contributions

We consider a set of n nodes, communicating over a synchronous network. That is, the protocol is executed in a sequence of *rounds* wherein in each round, a node can perform some local computation, send new messages to its neighbours, receive the messages sent to him in that round by the nodes (and if necessary perform some more local computation), in that order. The communication network is abstracted as an *undirected* graph. We further assume that there is a communication channel for each edge of the graph. Also, the communication channel between any two nodes is perfectly reliable and authenticated. We remark that all the nodes are *aware* of the topology of the network. During the execution, the adversary may corrupt up to t nodes. The adversary can make the corrupted node behave in an arbitrary way. Further, the adversary can read the internal states of up to another k nodes. We refer to such an adversary as (t,k)-adversary. One may view such an adversary as a mixed adversary.

We also assume the existence of authentication tools such as Public Key Infrastructure (PKI) and Digital Signature Schemes (DSS). Nodes can authenticate themselves and their messages with these authentication tools. We assume that every node has a secret key SK and a signature scheme $Sign(\mathsf{SK}, \mathsf{mesg})$ that allows it to sign the message mesg with its signature. Also, we assume that any

[1] Let $\gamma \in \{1, 3\}$ and define $\overline{\gamma}$ to be 3 if $\gamma = 1$ or 1 otherwise.

node can verify if a message carries the signature of a given node. It is assumed that the nodes sign whenever they send any message and also discard any received message that does not have a valid signature on it. This ensures that the receiver can uniquely identify the sender of the message. Since the adversary can look into the internal states of k nodes outside its control, it can forge the signature of all the k. So, in all the adversary can forge/generate the signatures of $(t + k)$ nodes. From now on, we use the term κ-connected network to denote a κ-vertex connected network. Also, throughout the paper we use n to denote the number of nodes in the network. Every node starts with an input value from the set $V = \{0, 1\}$.

Definition 1 (Byzantine Agreement)

- Agreement: *All non-faulty nodes decide on the same value* $u \in V$.
- Validity: *If all non-faulty nodes start with the same initial value* $v \in V$*, then* $u = v$.
- Termination: *All non-faulty nodes eventually decide.*

Here a node is considered as *faulty* if and only if he deviates from the delegated protocol. Therefore, the nodes that do not deviate from the designated protocol are *non-faulty* nodes. A node who follows the designated protocol diligently, even if adversary has complete access to his internal state is referred as *passively corrupt* node. A node is *honest* if he follows the designated protocol, and over whom adversary has absolutely no control. In particular, the adversary cannot replicate/forge the signature of honest nodes. For the purpose of this paper, we refer to both *honest* and *passively corrupt* nodes together as *non-faulty*.

Definition 2 ((t,k)-BA Protocol). *A protocol is a* (t, k)*-BA protocol if it accomplishes Byzantine agreement (ref. Definition 1) in the presence of a (t,k)-adversary.*

Definition 3 ((2t,t)-Connectivity). *A network* $\mathcal{N} = (\mathbb{P}, \mathcal{E})$ *is* $(2t, t)$*-Connected if its minimum degree is at least* $2t$ *and it is* $(t + 1)$*-connected.*

2.1 Results and Contributions

The contributions of this paper are many fold :

1. *Complete Characterization:* We give the necessary and sufficient condition(s) for designing protocols for agreement over (undirected) networks. We prove that BA Protocols over a n node network tolerating a (t,k)-adversary exist if and only if $n > 2t+ \min(t, k)$ and the network is

$$\begin{array}{rll} (t+1)\text{-connected} & if & n > (2t+k) \\ (2t,t)\text{-connected} & if\ (t+k) < & n \leq (2t+k) \\ (2t+1)\text{-connected} & if & n \leq (t+k) \end{array}$$

   The above holds for $k > 0$. For $k = 0$ it reduces to $n > t$ and network should be $(t + 1)$-connected.

2. *Unification:* In the standard authenticated model (ABA) [PSL80], the adversary can forge messages only on behalf of corrupt nodes. On the other hand, in the unauthenticated model (BA), every node can be treated as passively corrupt node[2]. Thus, characterizing possibility of BA protocols for the entire spectrum leads to unification of the extant literature on BA. As an elaboration consider the result presented in previous paragraph. It says if $n \leq (t+k)$ then $2t+1$ connectivity is necessary and sufficient – which is characterization of BA over non-authenticated setting. To elaborate, $n \leq (t+k)$ implies there are no honest nodes with a signature scheme that the adversary cannot forge and thus, collapses to the setting of non-authenticated BA.
3. *Agreement can be easier than all pair point-to-point communication:* From the results presented in this paper, it is evident that for all networks with $n > (2t+k)$, $(t+1)$-connectivity is sufficient for agreement. We, now, show that if $k > 0$ then for simulating point-to-point communication $(2t+1)$-connectivity is necessary. Consider a scenario where the Sender, say S, is non-faulty but the adversary can sign for S; in such a scenario, it is well-known that if the network is not $(2t+1)$-connected, there must exist a node j such that reliable message transmission from S to j is impossible[DDWY93]. From the above argument one can see that BA is easier than all pair point-to-point communication.
4. *BA is easier than Byzantine Generals (BG) [PSL80, LSP82]:* Informally, the problem of reliable broadcast in presence of Byzantine faults is also studied under the name of BG. Note that if a protocol for BG exists, then it vacuously is also a protocol for reliable point-to-point message transmission. Till this juncture, BA and BG have been isotopic forms. That is, BA iff BG. However, we show that BA and BG are two different problems and in fact BA is more primitive and fundamental to distributed computing than BG. In other words, there are several networks over which BA is possible whereas BG is impossible, in spite of having an overwhelming non-faulty majority.

Organization of the Proof. From now on, we assume that $k > 0$. We, also, assume that, *w.l.o.g*, either $n > (2t+k)$ or $(t+k) < n \leq (2t+k)$ or $n \leq (t+k)$. We consider these cases in Sections 3, 4 and 5 respectively. We establish the main theorem of the paper in Theorem 1.

3 The Good: When the Honest Are in Abundance

Lemma 1. *(t,k)-BA protocol over any general network $\mathcal{N} = (\mathbb{P}, \mathcal{E})$, $|\mathbb{P}| > (2t+k)$, exists if and only if $\mathcal{N}$ is $(t+1)$-connected.*

Proof. *Necessity:* The necessity of $(t+1)$-connectivity is straight forward due to the presence of t Byzantine faults. Elaborating further, the adversary may crash t nodes and disconnect the network.

[2] A node not having no authentication facility at all can also be visualized as passively corrupt and therefore, the adversary can forge messages on his behalf.

Sufficiency: We assume that every node i has a secret key $\mathsf{SK_i}$ and a signature scheme $Sign(\mathsf{SK_i}, \mathsf{mesg})$ that allows i to sign message mesg with i's signature. Also, we assume that any node can verify if a message carries a valid signature of i. It is assumed that the nodes sign whenever they send any message and also discard any received message that does not have a valid signature on it. Nodes run the Flood-Set protocol given in Algorithm 1.[3]

Algorithm 1. Flood-Set($\mathcal{N}, i, \sigma$) Node i starts with its input $\sigma \in \{0, 1\}$

```
$\Omega[n] = \emptyset$          ▷ Maintain a set for each node in the network, Initially empty
for each $j : (i, j) \in \mathcal{E}$ do
    $Send(\sigma, j)$                              ▷ $i$ sends its input to its neighbours
end for
$Round \leftarrow 1$
while $Round \leq 2n$ do                                     ▷ Flood for $2n$ rounds
    for each $j : (i, j) \in \mathcal{E}$ do
        $Receive(j)$
        $\forall x \in \mathbb{P},\ \Omega[x] = \Omega[x] \cup$ mesg's originating from node $x$ and received from $j$
    end for
    for each $j : (i, j) \in \mathcal{E}$ do
        $Send(\Omega, j)$                             ▷ Send messages to neighbours
    end for
    $Round = Round + 1$                                    ▷ Increment Round
end while
```

Node x deems the execution/invocation of the flood-set protocol by node j as *dirty* if he detects the Byzantine influence(i.e., if x received two different inputs from j with a valid signature of j or never receives any messages with valid signature of j); otherwise we say that the execution is *clean* for i. (Note that in our setting an execution can be *dirty* when either j is either faulty or passively corrupt.)

At the end of Flood-Set protocol, x modifies Ω in the following way: the j^{th} location of this tuple is changed to a $\perp$ if node j's flood-set execution is *dirty*, else if it is *clean* (and thus, had a single value v), then $\Omega[j] = v$. x takes a majority over Ω and outputs it as its decision value; otherwise, it outputs a default value. The proof of correctness hinges on the fact that $n > 2t + k$. Rearranging the terms, $(n - t - k) > t$, means that the honest nodes are in strict majority. Thus, the clean runs of the honest nodes carries us through. For a complete proof, we refer the readers to the full version [BGG+]. □

4 The Bad: When Faulty Outnumber the Honest

We now take a detour and limit our focus to 2-connected networks. Specifically, we first construct $(1, \psi)$-BA protocol on a 2-connected network, where $\psi \in [2, n - 2]$.

[3] This protocol is essentially the Dolev-Strong protocol [DS83] followed by n rounds of flooding.

Our approach can be outlined as follows: we design a protocol Π for the weakest case, that is to say the adversary *always* uses his full power and corrupts exactly 1 node actively and $(n-2)$ nodes do not have the power to authenticate themselves. It is straightforward to see that such a protocol would also work for the stronger assumptions in which more nodes have the power to authenticate themselves or the adversary does not corrupt anyone. Finally, we extend Π to a more general setting where the adversary controls $t \geq 1$ nodes.

4.1 Π: The Baby Protocol

Designing Π: Nodes exchange messages as per the Flood-Set protocol, given as Algorithm [1]. Node i applies a "modified" decision rule, which is as follows: If a majority exists over all the clean runs, then i outputs it as his decision and halts. Else, if the number of nodes which had a clean run is more than 2, i outputs 0 and halts. However, if the number of clean runs is only 2: Say, only runs of nodes a and b are clean. Notice that, one of the nodes a or b must be honest while the other node may be corrupt.

We now make a big assumption (and later show how to get rid of it): We assume that both a and b are non-faulty. Node x, $x \in \mathbb{P} - \{a, b\}$, sends his input to node a using the following routing strategy: (a) through the direct edge(if it exists), (b) otherwise a and b have at least 2 vertex disjoint paths between them and all the nodes in these paths must use these paths only. (c) else let $\chi = \{p_1, p_2 ..., p_j\}$ be the set of paths to a from x, if any such path includes b, x chooses it otherwise it chooses the shortest path to a. x also sends its input to b along an analogous set of rules. Nodes a and b are required to sign with their signature and send them back to x along the same path. If a and b receive more than one value from a node or not along the routing protocol given, a and b do not respond. Suppose, x receives, say, α and β from a and b respectively. If either a or b do not match with his input bit, x drops that message (Note that he cannot infer anything whether the node who signed on the toggled bit is corrupt or not).

Now, every node tries to get its input signed by a and b. After that, every node runs the Flood-Set protocol, given as Algorithm [1], twice. The first time with its input bit signed by a and the next time with its input signed by b. If both these runs turn out to be clean but the values contradict each other – both runs of i are overruled to be dirty. And if i has both its runs clean and consistent – then i's run is declared clean. At the end of Flood-Set protocol, i take a majority among all clean runs including the two runs of a and b; otherwise it decides on a default value(say, 0) and the protocol terminates. This would work as along as one of the node's execution in the flood-set protocol is clean. We, now, prove that if both a and b are non-faulty, then at least one node will have a clean run.

Lemma 2. *If both a and b are non-faulty, at least one node shall have a clean run.*

Proof. Since $\mathcal{N}$ is 2-connected, there can be two cases: (a) a, b are a vertex cut-set[4] in $\mathcal{N}$. (b) a, b are not a vertex cut-set. In the former, it is easy to see that

[4] A vertex cut-set in a network is a set of vertices whose removal from the graph makes it disconnected.

the claim is maintained as there shall be at-least two components upon removing a and b and it is clear that the adversary may be present only in one component. Thereby, the nodes in the other component can get the signature of a and b and hence, they will be able to sent their value to all other nodes successfully - one of a or b is honest and the other is non-faulty - so either both the runs will be clean with a consistent value or one of the runs will be clean and the other will be dirty - and from the protocol, both these cases are deemed to be clean runs. The case of both runs being clean with a contradictory value can happen only if one of a, b and the node is faulty.

In the later, there is only one component upon removing a and b. If a and b are adjacent, notice that the both a (resp. b) are sure to have a neighbor other than b (a) (as $\mathcal{N}$ is 2-connected, neighborhood of each node is at least 2). In this case, one of them is guaranteed to be non-faulty and hence from the routing method it is easy to see that, one of these node's will have a *clean* run (arguments go similar to previous case). If a and b are neither vertex cut-sets nor adjacent then there are at least 2 vertex disjoint paths between a and b. And active adversary resides in only one of them. Hence, at least of the nodes in the other path will send its input bit to get signed from a, b as per the routing algorithm. Hence, it easy to see that such a node *always* exists and hence at least one of the nodes modulo a, b will have a *clean* run. And, thus the Lemma. □

Observe that, all nodes would have agreed, under the big assumption that both a and b were non-faulty. However, if the protocol has not had the clean run - Lemma 2 implies that either a or b is faulty. Depending on whether nodes a and b are a vertex cut-set or not, nodes do the following.

The Shallow Side. If a and b are *not* a vertex cut-set, a publicly chosen (say, the node with the least UID outside a, b) non-faulty node i may send his input to everyone using paths outside a and b. Besides that, i sends his input to a through a path avoiding b and also, to b via path avoiding a. Note that these paths are bound to exist as it is a 2-connected network. This completes the construction of Π when a and b are not a cut-set. This simple protocol is a BA protocol as i is guaranteed to be a non-faulty node.

The Far Side: The Diamond Protocol. Consider the case when a and b are a vertex cut-set. Let a and b partition the network $\mathcal{N}$ into x components $c_1, c_2, \ldots c_x$. Nodes, now, choose a representative from each of these components, say n_i from is chosen from c_i (via some function on UID's). Nodes, now, create a (virtual) overlay-ed network $\mathcal{N}'$ whose vertices include n_i's, a and b. An edge appears between any two vertices only if there is an edge between the components represented by them. Note that we are not looking for a direct edge between n_i's, we are looking for edge between C_i's and a, b (each edge may have to be simulated via one or more edges). Notice that every n_i has to be connected to both a and b (follows from 2-connectivity of $\mathcal{N}$). Thus, $\mathcal{N}'$ (ref. Figure 2) has to be 2-connected (follows from $\mathcal{N}$ being 2-connected and that a, b are a cut). Now, we pair every node in $\mathcal{N}'$ (other than a and b) with another node (pairings are

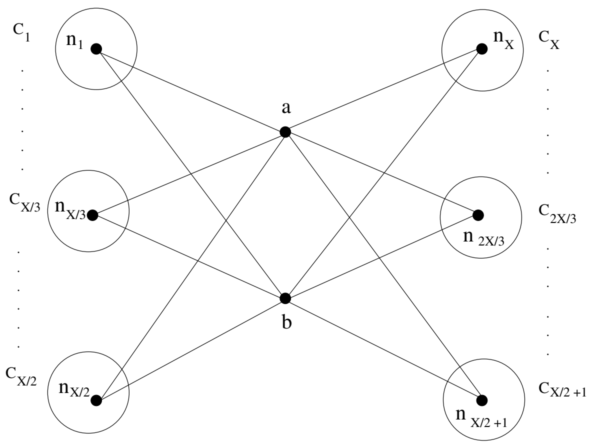


Fig. 2. Network $\mathcal{N}'$

chosen via a pre-decided strategy)[5]. Consider one such pairing (n_x, n_y). Now, nodes a, b, n_x and n_y simulate network G in Figure 1. Specifically, a, b, n_x and n_y simulate nodes 1, 3, 2 and 4 respectively. They execute rounds 1 and 2 of the Diamond protocol (Table 1). It is assumed that nodes sign on all the messages they send and discard any received message with an invalid signature. The rest of the nodes, merely, act as routers relying messages. The only difference w.r.t. the Diamond protocol is that in lieu of rounds 3 and 4, node 4(that is, n_y) executes the Flood-Set protocol given in Algorithm 1 on network $\mathcal{N}$ with the tuple containing inputs ψ_1 and ψ_3. If it is a *clean* run, nodes decide on that value and halt.

Claim. If node 4 does not receive the input value of node 2 by the end of this protocol, node 2 can identify the Byzantine faulty node.

Claim. Node 2 can sense if his input value has been reliably communicated to node 4.

The proofs of the above Claims have been omitted due to the constraints on space and can be obtained from [BGG+]. Notice that if nodes have not agreed on any value, we can invoke Claim 4.1 to infer that node 2(here it is, n_x) can identify the adversary (between a, b). If nodes haven't decided yet: nodes n_x and n_y swap their codes, that is, n_x deploys the code of 4 and vice-versa and then, they re-execute the Diamond protocol given above. If nodes still did not agree, both n_x and n_y can identify the faulty node is (out of a and b). The executions proceed until nodes have decided and halted or when all the pairings have tried their luck. When all the pairings are exhausted, notice that all n_i's can identify the adversary and this as good as the adversary making himself public!

[5] A node can be involved in multiple pairings.

Table 1. Diamond protocol

Code for node 1: Let node 1 start with an input $\sigma_1 \in \{0,1\}$. 1. Receives the values sent by nodes 2 and 4 and them across to 4 and 2 respectively. 2. Do nothing. 3. Receive ψ from node 4 and send it to 2. 4. Do nothing. 5. Receive from 2 and output the same.	**Code for node 3:** Let node 3 start with an input $\sigma_3 \in \{0,1\}$. 1. Receives the values sent by nodes 2 and 4 and them across to 4 and 2 respectively. 2. Do nothing. 3. Receive ψ from node 4 and send it to 2. 4. Do nothing. 5. Receive from 2 and output the same.
Code for node 2: Let node 2 start with an input $\sigma_2 \in \{0,1\}$. 1. Sends σ_2 to nodes 1 and 3. 2. Receive ψ_1 and ψ_3 from nodes 1 and 3 respectively. 3. Do nothing. 4. Receive ψ'_{31} from node 1 and ψ'_{13} from node 2. If any of these values is a $\perp$ then replace it with his input value, σ_2. If $\psi'_{13} = \psi'_{31} = \sigma_2$, then decide on σ_2. Else if $\psi_{ij} \neq \sigma_2$, then decide on ψ_j. 5. Send the value decided upon to 1 and 3.	**Code for node 4:** Let node 4 start with an input $\sigma_4 \in \{0,1\}$. 1. Sends σ_4 to nodes 1 and 3. 2. Receive ψ_1 and ψ_3 from node 1 and node 3 respectively. 3. Send the value ψ_3 to node 1 and ψ_1 to node 3. 4. Create a set W from ψ_1 and ψ_3. If $\|W\| = 1$ decide on that element, else output σ_4. 5. Do nothing.

All the n_i's agree on the input of the non-faulty node (out of a and b). Now, each of these n_i's send the decision to the nodes in the connected component represented by n_i and also to a and b. This completes the construction of Π ($(1, n-2)$-BA protocol).

Lemma 3. *For every* 2*-connected network on n nodes, Π is a* $(1, n-2)$*-BA protocol.*

Proof. Termination is obvious. For agreement, the use of Flood-Set (Algorithm 1) and the Diamond protocol for communication ensures that all the non-faulty nodes have consistent values and hence the decision rule simply implies that all of them *agree* on the same value. If all non-faulty nodes start with same input σ, every node's input (modulo the faulty ones) has to be σ and the protocol decides only upon receiving at least inputs from three nodes. Thus, if all the non-faulty nodes start with the same input, σ is the only possible output. Hence, by definition, it is a $(1, n-2)$-BA protocol over a 2-connected network. $\square$

4.2 Beyond 2-Connected Networks

We, now, extend Π to an arbitrarily connected network. Before that, we introduce new machinery. It is convenient to model the threshold adversary as a non-threshold adversary [HM97, FM98, HM00]. Informally, a non-threshold adversary captures the faults by a fault structure. That is, an enumeration of all the possible "snapshots" of faults in the network. Note that a single snapshot can be described by an ordered pair (B, K), where $B, K \subseteq \mathbb{P}$ and $B \cap K = \emptyset$,[6] which means that the nodes in the set B are Byzantine faulty while the nodes in the set K are passively corrupt. A fault structure is a collection of such pairs. More precisely, we define the fault structure by $\mathcal{A}$, where $\mathcal{A} \subseteq 2^{\mathbb{P} \times \mathbb{P}}$. The adversary is allowed to corrupt any pair from the fault structure. The fault structure is *monotone* in the sense that if $(B_1, K_1) \in \mathcal{A}$, then $\forall (B_2, K_2)$ such that $B_2 \subseteq B_1$ and $K_2 \subseteq K_1$, $(B_2, K_2) \in \mathcal{A}$. We note that $\mathcal{A}$ can be uniquely represented by listing the elements in its *maximal basis* $\overline{\mathcal{A}}$ which we define below. In what follows, unless specified otherwise, we work with only the maximal basis of $\mathcal{A}$.

Definition 4 (Maximal Basis of $\mathcal{A}$). *For any monotone fault structure $\mathcal{A}$, its maximal basis $\overline{\mathcal{A}}$ is defined as* $\overline{\mathcal{A}} = \{(B, K) | (B, K) \in \mathcal{A}, \nexists (X, Y) \in \mathcal{A}, (X, Y) \neq (B, K), X \supseteq B$ *and* $Y \supseteq K\}$.

It is evident that a (t, k)-fault is characterized by a fault basis $\mathcal{A} = \{(B, K) | |B| \leq t, |K| \leq k, B \cap K = \emptyset\}$. We define the size of the fault-basis to be the number of (B, K) pairs in the set $\mathcal{A}$. From now on, we work with the maximal basis of $\mathcal{A}$.

The Case of 3-Sized Structures. We first give the characterization for the case of 3-sized structures and then extend it to any adversary structure. We begin by setting the stage for constructing protocols on $\mathcal{N}$ tolerating $\mathcal{A}$. Since $(t + k) < n$ (and thus, $|B_i| + |K_i| < |\mathbb{P}|$), there is at least one honest node. Let us denote the honest node when the adversary corrupts (B_i, K_i) by h_i.

Assume that upon removing the nodes in $(B_1 \cup B_2 \cup B_3)$, say $\mathcal{N}$ is partitioned into x components, namely, $c_1, c_2, \ldots c_x$. Let us denote the honest node when the adversary corrupts (B_i, K_i) by h_i. Now, we choose a representative from each of B_1, B_2, B_3 and each of the components in the following fashion: If any of them have a h_i's, it is chosen as the representative; otherwise, the node with the lowest UID is picked. Note that, at most two of the three h_i's, say h_α and h_β, may lie within a B_i (this follows from the definition of h_i). Our goal is to ensure the presence of an honest representative. Consider the case when h_2 and h_3 lie inside B_1. In this case, if B_1 is corrupt, h_1 is honest and will have an honest representative and our target is achieved. However, when B_1 is not corrupt, one of h_2 or h_3 is honest (follows from definition of h_i's), but we are not sure which of them is honest. So, we need to be a little smarter in picking up a representative. Hence, we create a virtual node and use it as a representative in case both h_2 and h_3 lie inside B_1. The virtual node we create has the following property: Either it is honest or faulty but never passively corrupt. As with every virtual node, it is crucial to define its simulation, the notion of send/receive for the virtual node and its signature. For an exposition on virtual nodes, we refer the readers to [HM00].

Simulation of Virtual Node. Nodes h_2 and h_3 combine to simulate the virtual node. Since, they may not be adjacent, we need to specify how they

[6] This is not a serious assumption as if there some nodes common to both sets, such nodes can *w.l.o.g* placed solely in set B.

communicate. If h_2 and h_3 have a path consisting of nodes exclusively from B_1 and those outside the B_2 and B_3, they use this path to communicate and agree[7]. However, if all paths between h_2 and h_3 have a node either from B_2 or B_3, communication is carried out as follows: They send the values to each other via any two paths (chosen deterministically) such that one of them avoids nodes in B_2 and the other avoids B_3(such paths are guaranteed to exist as the network is $(t+1)$-connected). h_2 and h_3, now, take a majority over these values among the values obtained in the *clean* runs[8]. If they share a path exclusively in B_1 - when B_1 is not corrupt, the objective of creating an honest is achieved as h_2, h_3 are non-faulty and they share a good path and when it is corrupt, the simulation is allowed to fail. When the construction uses 2 paths (one from B_2 and the other from B_3), if either h_2 or h_3 has a signature scheme which the adversary cannot replicate (in other words, honest) - the only value that can be received consistently is the value of the honest node. Thus, the simulation is consistent. While if one of them is faulty, the protocol does not rely on the simulation and we are allowed to fail.

Signature. The notion of signature for a virtual node is a natural extension of the simulation. Any message which has to be signed by the virtual node contains a sequence of signatures from h_2, h_3 and the nodes along the communication path. The verification relies on the fact that every node knows h_2, h_3 and the nodes involved in the simulation. A valid signature amounts to a correct sequence of signatures on the message sent by the virtual node.

Send/Receive. Virtual node sending a message to node i is defined as follows: h_2 and h_3 exchange messages using the aforementioned paths and then run the Flood-Set protocol. Now they send the transcripts to i. i takes a majority among the *clean* runs of the Flood-Set reveals the transcript. Once the transcript is received, the message of the virtual node is extracted. The Flood-set works when the virtual node is honest. If it is not honest, the flooding is allowed to fail. i sending a message to the virtual node is equivalent to - i sending message separately to h_2 and h_3 and they exchange the messages they received from i and then, take a majority over the clean runs. The consistency of send, receive rely on the paths chosen and the fact that one of the paths is always good. The signatures of the nodes along the path play a crucial role in allowing i to verify the transcripts.

Lemma 4. *(t,k)-BA protocol over any general network $\mathcal{N} = (\mathbb{P}, \mathcal{E})$, when $(t+k) < |\mathbb{P}| \leq (2t+k)$, tolerating a 3-sized fault basis $\mathcal{A} = \{(B_1, K_1),\ (B_2, K_2),\ (B_3, K_3)\}$, $\forall (B_i, K_i) \in \mathcal{A}$, $|B_i| = t$ and $|K_i| = k$, exists if and only if $\mathcal{N}$ is $(2t, t)$-connected.*

[7] Which is only possible when both are non-faulty. However, when they are a faulty they cannot agree and the simulation fails. However, in this case the protocol does not rely on this virtual node to provide a honest representative.

[8] Clean runs are those in which only one message with a valid signature is received.

Proof. Sufficiency: Say n_i is chosen from c_i (components formed after removing B_1, B_2 and B_3 from $\mathcal{N}$) and b_i from $B_i, i \in \{1, 2, 3\}$. We, now, create a overlay-ed network $\mathcal{N}'$(this construction is similar to the section on diamond protocol) on n_i's and b_i's in which an edge appears between any two nodes(representatives) only if there is a edge between the components represented by them. Each of these n_i's is connected to at least two b_i's (Since, $\mathcal{N}$ is $(t + 1)$-connected).

Notice that $\mathcal{N}'$ is 2-connected (similar to the argument in The far side in Section 4.1). Also, atleast one of the representatives is honest (follows from the election of representatives). Hence, nodes in $\mathcal{N}'$ now execute Π (ref. Section 4.1) and by Lemma 3 all the (non-faulty) representatives agree. After the representatives agree, they distribute their decision across their components and to the representatives of B_i's. For nodes that lie inside B_i's: Since the network $\mathcal{N}$ is $(2t, t)$-connected (Definition 2), every node has a degree at least $2t$. This implies that any node in B_i has a direct edge to a node outside all B_i's OR has a direct edge to one node from each of the B_i's. In the former, it can decide on the value obtained from outside B_i's. In the latter, it takes a majority among the three values received from each of the B_i's. This is bound to work as two of the three B_x's are non-faulty and would have agreed in the execution of Π. The proof of correctness stems from the fact that once the representatives agree (similar to Lemma 3), it is easy that all the nodes in their respective components also agree. *Necessity:* We construct two executions and prove that there is a non-faulty node for which both of them are indistinguishable. Thus, the node remains in a bivalent state forever. It is an argument along the lines of canonical impossibility proofs in distributed computing. For further details, one may refer to the full version of the paper [BGG+]. □

3-Sized to n-Sized. We, now, extend the characterizations over a 3-sized fault basis to an n-sized fault basis.

Lemma 5. *(t, k)-BA protocol over a network $\mathcal{N}$ tolerating a n-sized fault basis $\mathcal{A}$ exists if and only if there exists (t, k)-BA protocols for every 3-sized fault basis $\mathcal{B}$, $\mathcal{B} \subseteq \mathcal{A}$.*

This extension is along the lines of [HM00] and readers may refer to the full version of the paper [BGG+].

Lemma 6. *(t, k)-BA protocol over a network $\mathcal{N} = (\mathbb{P}, \mathcal{E})$, $(t+k) < n \leq (2t+k)$, exists if and only if $\mathcal{N}$ is $(2t, t)$-Connected.*

Proof. By invoking Lemma 4 and Lemma 5. □

5 The Ugly: When Only the Faulty can Authenticate

Lemma 7. *(t, k)-BA Protocol over $\mathcal{N}$, $n \leq (t + k)$, exists if and only if $\mathcal{N}$ is $(2t + 1)$-connected.*

Proof. Since $n \leq (t+k)$, it basically means that the signatures schemes of all nodes outside adversary's control can be forged and hence the power of authentication is entailed useless. Hence, proofs from the standard unauthenticated model of BA [Lyn96, Dol82] will lead us here. □

Main Theorem 1. *(t,k)-BA protocol over a network $\mathcal{N} = (\mathbb{P}, \mathcal{E})$, $|\mathbb{P}| = n$, exists if and only if $n > 2t+$ min(t,k) and $\mathcal{N}$ is*

$$\begin{array}{rlr} (t+1)\text{-}connected & if & n > (2t+k) \\ (2t,t)\text{-}connected & if\ (t+k) < & n \leq (2t+k) \\ (2t+1)\text{-}connected & if & n \leq (t+k) \end{array}$$

Proof. By invoking Lemma 1, Lemma 6 and Lemma 7 and the result of Gupta *et al.* [GGBS10], we establish the theorem. □

6 Concluding Remarks

Possibly, for the first time in literature we show that there are networks over which agreement is possible even though not all non-faulty nodes can reliably communicate with each other. In essence, *all-node global consistency is strictly easier than all-pairs point-to-point communication.* In this perspective, it appears that the problem of *agreement* could be a more fundamental primitive to general distributed computing than what (even the ubiquitous problem of) reliable communication is.

The focus of this work has been, primarily, to establish (im)possibility results for BA. Hence, the protocols presented in this paper are sub-optimal and there is a definite scope for improving the same. Further, it will be interesting to study this problem in more generic settings such as directed networks, asynchronous networks and may be under the influence of a non-threshold adversary.

References

[AFM99] Altmann, B., Fitzi, M., Maurer, U.M.: Byzantine agreement secure against general adversaries in the dual failure model. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 123–139. Springer, Heidelberg (1999)

[BGG$^+$] Bansal, P., Gopal, P., Gupta, A., Srinathan, K., Vasishta, P.K.: Byzantine agreement using partial authentication. Technical report, `http://people.csail.mit.edu/prasant/agreement.pdf`

[Bor95] Borcherding, M.: On the number of authenticated rounds in byzantine agreement. In: Helary, J.-M., Raynal, M. (eds.) WDAG 1995. LNCS, vol. 972, pp. 230–241. Springer, Heidelberg (1995)

[Bor96a] Borcherding, M.: Levels of authentication in distributed agreement. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 40–55. Springer, Heidelberg (1996)

[Bor96b] Borcherding, M.: Partially authenticated algorithms for byzantine agreement. In: ISCA: Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, pp. 8–11 (1996)

[DDS87] Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. J. ACM 34(1), 77–97 (1987)

[DDWY93] Dolev, D., Dwork, C., Waarts, O., Yung, M.: Perfectly Secure Message Transmission. Journal of the Association for Computing Machinery (JACM) 40(1), 17–47 (1993)

[Dol82] Dolev, D.: The Byzantine Generals Strike Again. Journal of Algorithms 3(1), 14–30 (1982)

[DS83] Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM Journal on Computing 12(4), 656–666 (1983)

[FLP85] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)

[FM98] Fitzi, M., Maurer, U.M.: Efficient byzantine agreement secure against general adversaries. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 134–148. Springer, Heidelberg (1998)

[GGBS10] Gupta, A., Gopal, P., Bansal, P., Srinathan, K.: Authenticated Byzantine Generals in Dual Failure Model. In: Kant, K., Pemmaraju, S.V., Sivalingam, K.M., Wu, J. (eds.) ICDCN 2010. LNCS, vol. 5935, pp. 79–91. Springer, Heidelberg (2010)

[GLR95] Gong, L., Lincoln, P., Rushby, J.: Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults (1995)

[HM97] Hirt, M., Maurer, U.: Complete Characterization of Adversaries Tolerable in Secure Multi-party Computation. In: Proceedings of the 16th Symposium on Principles of Distributed Computing (PODC), August 1997, pp. 25–34. ACM Press, New York (1997)

[HM00] Hirt, M., Maurer, U.M.: Player simulation and general adversary structures in perfect multiparty computation. J. Cryptology 13(1), 31–60 (2000)

[LSP82] Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)

[Lyn96] Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Mateo (1996)

[PSL80] Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)

[ST87] Srikanth, T.K., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing 2(2), 80–94 (1987)

On Approximate Distance Labels and Routing Schemes with Affine Stretch

Ittai Abraham[1] and Cyril Gavoille[2,⋆]

[1] Microsoft Research, Silicon Valley Center, USA
ittaia@microsoft.com
[2] Université de Bordeaux, LaBRI, France
gavoille@labri.fr

Abstract. For every integral parameter $k > 1$, given an unweighted graph G, we construct in polynomial time, for each vertex u, a distance label $L(u)$ of size $\tilde{O}(n^{2/(2k-1)})$. For any $u, v \in G$, given $L(u), L(v)$ we can return in time $O(k)$ an *affine* approximation $\hat{d}(u,v)$ on the distance $d(u,v)$ between u and v in G such that $d(u,v) \leqslant \hat{d}(u,v) \leqslant (2k-2)d(u,v) + 1$. Hence we say that our distance label scheme has affine stretch of $(2k-2)d + 1$. For $k = 2$ our construction is comparable to the $O(n^{5/3})$ size, $2d + 1$ affine stretch of the distance oracle of Pătraşcu and Roditty (FOCS '10), it incurs a $o(\log n)$ storage overhead while providing the benefits of a distance label. For any $k > 1$, given a restriction of $o(n^{1+1/(k-1)})$ on the total size of the data structure, our construction provides distance labels with affine stretch of $(2k-2)d+1$ which is better than the stretch $(2k-1)d$ scheme of Thorup and Zwick (J. ACM '05). Our second contribution is a compact routing scheme with poly-logarithmic addresses that provides affine stretch guarantees. With $\tilde{O}(n^{3/(3k-2)})$-bit routing tables we obtain affine stretch of $(4k-6)d + 1$, for any $k > 1$. Given a restriction of $o(n^{1/(k-1)})$ on the table size, our routing scheme provides affine stretch which is better than the stretch $(4k-5)d$ routing scheme of Thorup and Zwick (SPAA '01).

1 Introduction

A *distance label* scheme is a pair of protocols. A *pre-processing* protocol takes a graph as input and maps each vertex u to a label $L(u)$ and a *query* protocol that takes two labels $L(u), L(v)$ as input and outputs an estimate $\hat{d}(u,v)$. Typically, distance label schemes are measured by (1) the time complexity of the prepossessing protocol, (2) the size of the labels, (3) the time complexity of the query algorithm, and (4) the quality of the distance estimation. Thorup and Zwick [17] prove that for any integer $k > 1$ it is possible to preprocess a graph in polynomial time to obtain labels of size at most[1] $\tilde{O}(kn^{1/k})$. Given any $L(u), L(v)$,

⋆ Also member of "Institut Universitaire de France". Supported by the ANR-project "ALADDIN", and équipe-projet commune LaBRI-INRIA Bordeaux Sud-Ouest "CÉPAGE". This work has been partially done during a visit at Microsoft in April 2011.

[1] Tilde-O notation are similar to O notation up to factors poly-logarithmic in n.

in time $O(k)$, a distance estimation $d(u,v) \leqslant \hat{d}(u,v) \leqslant (2k-1)d(u,v)$ is provided. Distance labels are a special type of a *Distance Oracle*, a global data structure that returns estimations on the all-pairs distance matrix. Distance labels have the benefit of allowing to easily partition the distance oracle into sub-regions. For example, suppose we have a distance label scheme for the whole world road network, it is easy to distribute to each mobile phone in a succinct manner the part of the map in the world that is relevant to that particular phone.

In this paper we focus on unweighted graphs and consider distance estimations $\hat{d}$ that have both a linear multiplicative term and an additive term. We say that a distance estimation $\hat{d}$ has *affine stretch* of $\alpha d+\beta$ if $d(u,v) \leqslant \hat{d}(u,v) \leqslant \alpha d(u,v)+\beta$ for all u,v. It is known that any distance oracle of $o(n^2)$ size must have affine stretch $\alpha d + \beta$ such that $\alpha + \beta \geqslant 3$, and this is true even if we restrict our attention to unweighted graphs (see [17]).

Pătraşcu and Roditty [14] prove that unweighted graphs have a distance oracle of $O(n^{5/3})$ expected size that provides affine stretch of $2d+1$. Given a restriction of $o(n^2)$ on the size of the distance oracle, the $2d+1$ affine stretch of [14] is better than the stretch $3d$ distance oracle of [17]. (The [17] scheme uses only $O(n^{3/2})$ memory). However, the scheme of [14] seems to require a global data structure. We could not see an obvious way to distribute this information into balanced labels of size $\tilde{O}(n^{2/3})$.

Our first contribution is a distance label scheme that for any integer $k > 1$, can be pre-processed in polynomial time and produce labels of size $\tilde{O}(n^{2/(2k-1)})$ that provide distance estimation in $O(k)$ time with affine stretch of $(2k-2)d+1$. Relative to the $2d+1$ result of [14], our $k=2$ scheme requires $O(\log^{2/3} n)$ more total memory but provides the benefits of a distance label. Note that for any $k > 1$, our $(2k-2)d+1$ scheme requires more memory than the [17] $(2k-1)d$ scheme (but has better stretch) and less memory than the [17] $(2k-3)d$ scheme (but has worse stretch). To highlight our contribution relative to the best previous results, consider a problem where there is some external restriction of $o(n^{1+1/(k-1)})$ on the total size of the distance oracle, and ask for the best affine stretch under such restrictions. Our construction provides distance labels with affine stretch of $(2k-2)d+1$ which is better than the best previous solution which obtains stretch $(2k-1)d$. Note that according to an Erdös-Simonovits conjecture about density of large girth graphs [8], any distance oracle on unweighted graphs with $o(n^{1+1/(k-1)})$ memory must have an affine stretch $\alpha d+\beta$ with $\alpha+\beta \geqslant 2k-1$.

Technically, we make two contributions. First we observe that the scheme of [14] naturally[2] generalizes to a distance oracle with $O(n^{1+2/(2k-1)})$ memory and affine stretch of $(2k-2)d+1$ using the hierarchical sampling scheme of [17]. The main technical contribution of [14] was a new ball growing protocol. Our second contribution it to show that for distance labels one can instead use the sampling technique of [16] to obtain similar bounds.

For stretch $2d+1$ with $\tilde{O}(n^{2/3})$ labels the high level idea is to build clusters $B(u)$ around each node u such that $|B(u)| = \tilde{O}(n^{1/3})$ and for every $w \in B(u)$,

[2] We have learnt that M. Patrascu, L. Roditty, M. Thorup have independently made a similar observation.

$|\{v : w \in B(v)\}| = \tilde{O}(n^{1/3})$. These clusters are built using the sampling technique of [16]. Given two nodes s and t the proof then proceeds much like [1,14], by separating into two cases. If $B(s) \cap B(t) \neq \varnothing$ then we can get the exact distance. Otherwise if $B(s)$ and $B(t)$ are disjoint, then we use the observation of [14] that the diameter of the smallest of the two balls is at most $(d(s,t)+1)/2$.

In the second part we study compact routing schemes with affine stretch. A routing scheme maps each vertex to some routing information (its routing table) and a poly-log size label (its address). Given a target label, the source needs to decide with its routing table how to forward a message using a poly-log sized header. The affine stretch of a routing scheme is a bound on the worst case route length taken by the routing scheme relative to the shortest path. Routing schemes are more challenging than distance labels since the source has access only to a poly-log size label of the target (while in the distance label model we had symmetric information about the source and target). Intuitively, this asymmetry generates difficulties in maintaining similar space-stretch trade-off in comparison with distance labeling or distance oracles. On the other hand, when progressing to the target, a message can profit from vertices it traverses. Nevertheless, for small distances in a sparse graph this advantage seems negligible. Given space bound of $\tilde{O}(n^{1/k})$ and $k > 2$, there are gaps between the best known distance labeling and the best known compact routing. Distance labeling achieve stretch $2k-1$ [17], whereas the best known routing schemes only achieves stretch $4k-5$ [16].

Our second contribution is a compact routing scheme that with $\tilde{O}(n^{3/(3k-2)})$-bit routing tables obtains affine stretch of $(4k-6)d+1$, for any $k > 1$. Our $(4k-6)d+1$ scheme requires more memory than the [16] $(4k-5)d$ scheme (but has better stretch) and less memory than the [16] $(4k-9)d$ scheme (but uses less memory). To highlight our contribution relative to the best previous results, consider a problem where there is some external restriction of $o(n^{1/(k-1)})$ on the size of the routing tables, and ask for the best affine stretch under such restrictions. Our construction provides a compact routing scheme with affine stretch of $(4k-6)d+1$ which is better than the best previous solution which obtains stretch $(4k-5)d$ (see [16]).

Our stretch $2d+1$ routing scheme is based on our approach for $2d+1$ labeling, but requires some additional ideas. The main new difficulty is when the smaller ball is around the source. This requires to redefine the ball around the source, spread routing information among vertices, and using hashing to find the relevant information. Such ideas were previously used (to the best of our knowledge) only for name-independent routing (for example, see [1]). For the general case we need an additional step. When the smaller ball is around the source we first try routing near the ball to gather information but if this fails we go back to the source and then proceed as in [16].

Related Work. One idea to reduce the size of the representation of distance information is to use sparse graph spanners, that is a spanning subgraph of the original graph using possibly less edges while preserving distances between vertices up to some estimation. A simple extension of the Kruskhal's greedy algorithm [3] shows that, for every $k \geqslant 1$, every weighted graph with n vertices has a spanner of size $O(n^{1+1/k})$ and stretch $(2k-1)d$. As quoted in the introduction, according to some girth conjecture, there are unweighted graphs on which every stretch $\alpha d + \beta$ spanner has size $\Omega(n^{1+1/k})$ if $\alpha + \beta < 2k+1$ (this is proved for $k = 1, 2, 3, 5$, and for all k if $\alpha = 1$ [19]). In the last decade many constructions have been obtained for different trade-offs between α and β, see for instance [5,6,7,13,18]. It seems that the picture of possible trade-offs is far from complete.

Graph spanners give a trivial compact data structure to represent approximate distances, however they do not give a fast way to extract approximate distances or to extract the best path in the spanner. Typically, sparse graphs (say with $o(n^{1+1/k})$ edges) have trivial spanners (the graph itself), but extracting short paths in the spanner is as hard as extracting shortest paths in the original graph. So, other more "structured" constructions of spanners have been given, based on tree-covers or partitions [4,12,17], that support fast approximate distance queries, and so lead to compact and fast distance oracles. They achieve space $\tilde{O}(n^{1+1/k})$, affine stretch $O(kd)$, and query time $O(k)$ (and even time $O(1)$ for [12]). Interestingly, [15] have showed that every distance oracle on sparse graphs that supports time t stretch αd distance queries must have space $n^{1+\Omega(1/(\alpha t))}$. So, the space bound of a distance oracle is constrained not only by the representation of approximate distances (or spanner representation) but also by the query time: fast distance oracles imply large space data structures, independent of the density of the input graph. In this context [14] have showed that, under a conjecture about set intersecting data structures, any distance oracle on sparse unweighted graphs of $\tilde{O}(n)$ edges with affine stretch $< 2d+1$ requires space $\Omega(n^{1.5})$.

Compact Routing has been already investigated in the late '70s for the very first interconnected computer networks [11]. Trade-offs between the size of the routing tables and the stretch on the route length are similar to the one achieved by spanners and distance oracles. Many results have been publish in this fields in the last decade, in particular for routing in specific graph classes (low dimension networks, scale free networks, planar networks, road networks, and so on), capturing the topology of real networks. For general graphs, the state-of-the-art is the Thorup and Zwick routing scheme [16] that, with $\tilde{O}(n^{1/k})$ routing tables and polylog addresses, routes along paths with affine stretch $(4k-5)d$. Like distance oracles, under a girth conjecture, the lower bound on the stretch is $\alpha + \beta \geqslant 2k-1$ for any stretch $\alpha d + \beta$ routing scheme of space $o(n^{1/(k-1)})$. So, optimal routing schemes is known only for $k = 1$ and 2. If we consider routing schemes with affine stretch $d + \beta$, then a lower bound of $\Omega(n/\beta^2)$ on the space exists [10]. This latter lower bound indicates, even if girth and set intersecting conjecture do not hold, that trade-offs in compact routing are different from those for spanners. While graph spanners of $o(n)$ edge density exist for affine stretch $d+2$ ([2]) and $d+6$ ([5]), affine stretch $d+\beta$, for *any* constant β, cannot be guaranteed by a $o(n)$ memory routing scheme even with arbitrarily large routing decision time.

2 Fast Approximate Distance Labeling

Theorem 1. *Let $k \geqslant 2$ be an integer. Every unweighted n-vertex graph enjoys a distance labeling with affine stretch $s(d) = (2k-3)d + 2\lceil d/2 \rceil \leqslant (2k-2)d+1$, labels of length $\tilde{O}(n^{2/(2k-1)})$, and query time $O(k)$. Construction of the labels takes polynomial time.*

Hereafter, our constructions are described as randomized. Most of randomness comes from the construction of some small hitting sets, or vertex coloring, for which deterministic versions of polynomial time complexity exist.

For the sake of the presentation, we will sketch the first case of our distance labeling, whenever $k = 2$. It achieves stretch $d + 2\lceil d/2 \rceil \leqslant 2d+1$ and labels of length $\tilde{O}(n^{2/3})$. Most of the ideas of this base case will be reused for the general construction, and also for routing.

Let $G = (V, E)$ be any unweighted graph with n vertices. The construction has some similarities with the technique used in the Thorup-Zwick compact routing scheme [16]. It is based on the selection of some subset of vertices $L \subseteq V$ named hereafter *landmarks*.

Given a set of landmarks L and a base set $W \subseteq V$, we define, for every vertex $u \in V$, its *ball* with respect to W and L as: $B_{W,L}(u) = \{v \in W : d(u,v) < d(u,L)\}$ where $d(u,L) = \min\{d(u,\ell) : \ell \in L\}$. In other words, $B_{W,L}(u)$ consists of all the closest vertices of W around u up to the closest landmark. For every $v \in W$, we define $C_{W,L}(v) = \{u \in V : v \in B_{W,L}(u)\}$.

We slightly generalize Theorem 3.1 of Thorup-Zwick [16] (proof in full version):

Lemma 1. *Given a graph $G = (V, E)$, size parameter s and base set $W \subseteq V$, one can construct in polynomial time a landmark set L such that for every vertex u of G, $|B_{W,L}(u)| \leqslant 4|W|/s$, $|C_{W,L}(u)| \leqslant 4|V|/s$, and, in expectation, $|L| \leqslant 4s \log |V|$.*

The key property of Lemma 1 is that *all* $C_{W,L}$'s balls are bounded by $O(|V|/s)$, and not only in expectation. In the remaining of this section, we will choose $W = V$, and for convenience, we will drop subscript V from B's and C's balls. So, $B_L(u) = B_{V,L}(u)$, and $C_L(v) = C_{V,L}(u)$. An important observation is that $u \in C_L(v)$ if and only if $v \in B_L(u)$.

We apply Lemma 1, so with $W = V$, and with $s = (n^2/\log n)^{1/3}$, so that $|L| = O((n \log n)^{2/3})$ and $|B_L(u)|, |C_L(u)| = O((n \log n)^{1/3})$.

Storage for Vertex u. It stores in its labels the set of vertices:

$$I(u) \;=\; L \cup B_L(u) \cup \left(\bigcup_{v \in B_L(u)} C_L(v) \right) .$$

It also stores the distances from u to each vertex $v \in I(u)$, and its closest landmark, say $\ell(u)$ (breaking ties arbitrary). We easily check that $|I(u)| = O((n \log n)^{2/3})$. Thus the label length is $O(n^{2/3} \log^{5/3} n)$ bits. The labeling scheme is clearly polynomially constructible.

Querying between s and t.

If $t \in I(s)$, then returns $d(s,t)$, else
returns $\min\{d(s,\ell(s)) + d(\ell(s),t), d(t,\ell(t)) + d(\ell(t),s)\}$.

The query can be solved using the labels of s and t only. It takes constant time to determine if $t \in I(s)$ using linear size and worst-case constant time static membership data-structures. Then, a linear size hashing tables can extract from $I(s)$ in constant time $d(s,w)$ for any vertex $w \in I(s)$ (see [17] for further details). So answering query (s,t) is done in constant time.

Stretch Analysis. If $t \in I(s)$, then the returned value is indeed the distance $d(s,t)$. Assume $t \notin I(s)$. We observe that $B_L(s)$ and $B_L(t)$ must be disjoint in that case. If not, say $v \in B_L(s) \cap B_L(t)$, then $v \in B_L(t)$ implies $t \in C_L(v)$. Since $v \in B_L(s)$ too, then $C_L(v) \subset I(s)$. Therefore, $t \in I(s)$: a contradiction.

Let $d = d(s,t)$, and let $\hat{d} = \min\{d(s,\ell(s)) + d(\ell(s),t), d(t,\ell(t)) + d(\ell(t),s)\}$ be the value returned by the oracle. Without loss of generality, assume that $d(s,\ell(s)) \leqslant d(t,\ell(t))$. Using the triangle inequality between $\ell(s)$ and t, we have:

$$\hat{d} \leqslant d(s,\ell(s)) + d(\ell(s),t) \ \leqslant\ d(s,\ell(s)) + (d(\ell(s),s) + d(s,t)) = 2d(s,\ell(s)) + d\ .$$

Consider a shortest path P from s to t. Let $x \in P \cap B_L(s)$ be the farthest from s, and, similarly, let $y \in P \cap B_L(t)$ be the farthest from t. Note that, since $B_L(s)$ and $B_L(t)$ are disjoint, $d(x,y) \geqslant 1$. Because x, y are on P, we have:

$$d \ =\ d(s,x) + d(x,y) + d(y,t) \ \geqslant\ d(s,x) + d(y,t) + 1\ . \tag{1}$$

By definition of x, the neighbor of x on P at distance $d(s,x)+1$ from s is not in $B_L(s)$. So its distance to s is $d(s,x)+1 \geqslant d(s,\ell(s))$. The same argument applies to y and t, so that $d(t,y)+1 \geqslant d(t,\ell(t))$.

Plugging in Eq. (1), and combining with the assumption $d(s,\ell(s)) \leqslant d(t,\ell(t))$, we obtain:

$$d \ \geqslant\ d(s,\ell(s)) + d(t,\ell(t)) - 1 \ \geqslant\ 2d(s,\ell(s)) - 1$$

In other words, $d(s,\ell(s)) \leqslant \lfloor (d+1)/2 \rfloor = \lceil d/2 \rceil$. Since, $\hat{d} \leqslant 2d(s,\ell(s)) + d$, we have proved that $\hat{d} \leqslant d + 2\lceil d/2 \rceil$.

General Construction: $k \geqslant 2$. For the general case, we will make the use of *k levels* of landmark sets: $L_0, L_1, \ldots, L_{k-1}$ where $L_0 = V$ for convenience. We denote by $\ell_i(u)$ the closest landmark from u in L_i (breaking ties arbitrary). Note that $\ell_0(u) = u$. The notations $\ell(u)$ and L for case $k = 2$ simply denotes here $\ell_1(u)$ and L_1 respectively.

The collection of landmark sets is constructed by applying $k-1$ times Lemma 1. The first time, we apply it with base set $W_1 = L_0 = V$ and size parameter $s_1 = n^{1-1/(2k-1)}$ so $|B_{L_1}(u)|, |C_{L_1}(u)| = O(n/s_1) = \tilde{O}(n^{1/(2k-1)})$. Then, for each $i \in [2, k-1]$, we fix the base set $W_i = L_{i-1}$ and $s_i = n^{1-(2i-1)/(2k-1)}$. Note that $s_{k-1} = n^{2/(2k-1)}$, and that $s_{i-1}/s_i = n^{2/(2k-1)}$ for all $i \geqslant 1$.

Storage for Vertex u. It stores in its labels the set of vertices:

$$I(u) \;=\; L_{k-1} \cup \left(\bigcup_{i=1}^{k-1} B_{L_{i-1},L_i}(u) \right) \cup \left(\bigcup_{v \in B_{L_1}(u)} C_{L_1}(v) \right) \;.$$

It also stores in its label the distances from u to each vertex $v \in I(u)$, and its sequence of closest landmarks $\ell_0(u), \ldots, \ell_{k-1}(u)$. Note that the distance to each landmark $\ell_i(u)$ is already in the label of u.

Applying Lemma 1, the size of $I(u)$ is (details in the full version): $|I(u)| = \tilde{O}(s_{k-1}) + \tilde{O}\left(\sum_{i=2}^{k-1}(s_{i-1}/s_i)\right) + O(n/s_1)^2 = \tilde{O}(n^{2/(2k-1)})$.

Querying between s and t.

1. If $d(s,\ell_1(s)) > d(t,\ell_1(t))$, exchange the role of s and t in the next two steps.
2. Compute the smallest index i_0 such that $\ell_{2i_0}(t) \in I(s)$ or $\ell_{2i_0+1}(s) \in I(t)$.
3. If $\ell_{2i_0}(t) \in I(s)$ returns $d(s,\ell_{2i_0}(t)) + d(\ell_{2i_0}(t),t)$, else returns $d(s,\ell_{2i_0+1}(s)) + d(\ell_{2i_0+1}(s),t)$.

Intuitively, the answering algorithm tries to approximate the distance successively thanks to the sequence of landmarks $\ell_0(t), \ell_1(s), \ell_2(t), \ell_3(s), \ldots$, respectively with the paths $s \to \ell_i(t) \to t$ for even i, and $s \to \ell_i(s) \to t$ for odd i. The query can be solved using the labels of s and t only. It takes $O(k)$ to determine i_0 using static dictionary. Then, it takes constant time to return the distance using hash table.

Stretch Analysis for $k \geqslant 2$. Appears in the full version.

3 Compact Routing

Theorem 2. *Given an unweighted connected graph with n vertices and an integral parameter $k \geqslant 2$, there exists a polynomial algorithm that produces a labeled routing scheme which requires $\tilde{O}(n^{3/(3k-2)})$-bit routing tables per vertex, $o(k\log^2 n)$ labels, and $o(\log^2 n)$ headers, that performs routing decisions in $O(k)$ time and routes along paths of affine stretch at most $(4k-7)d + 2\lceil d/2 \rceil \leqslant (4k-6)d + 1$.*

To prove Theorem 2, we need to introduce a third kind of ball, defined by volume. For a vertex u, set L, and parameter t, let $E_L(u,t)$ be the subset of L that consists of the t closest vertices to u (breaking ties with any uniform policy). We can check that B, C, E balls share the following monotonicity property which is important for routing:

Property 1. Let $X(\cdot)$ denote one type of ball among $B_L(\cdot)$, $C_L(\cdot)$, and $E_L(\cdot,t)$, for a given set $L \subseteq V$ and parameter t. Then, if $t \in X(s)$ and u is on a shortest path from s to t, then $t \in X(u)$.

A key technique in the proof of Theorem 2 is to spread routing information using a commonly known hash function. This idea is standard in *Name-Independent* routing (see for example [1]). To the best of our knowledge this is the first use of such technique for *Labeled* routing (where vertices are given poly-log size labels).

We also make use of the following labeled routing scheme for trees:

Lemma 2. **[9,16]** *Every spanning tree T of a graph with n vertices has a labeled routing scheme that, given any destination label, routes optimally on T from any source to the destination. The storage per vertex, the label size, and the header size are $O(\log^2 n/\log\log n)$ bits. Given the stored information of a vertex and the label of the destination, routing decisions take constant time.*

For a tree T containing a vertex v, let $\mu(T,v)$ denote the routing information stored at vertex v and $\lambda(T,v)$ denote the destination label of v in T as defined by the labeled routing scheme of Lemma 2.

For the sake of the presentation, we present first the basic construction for $k=2$. Then, we present the construction for $k=3$ which introduces new tools needed for $k>2$.

Proof of Theorem 2 for $k=2$. The routing tables in this case have size $\tilde{O}(n^{3/4})$, and the routing scheme has affine stretch $d+2\lceil d/2\rceil \leqslant 2d+1$.

Let L be a set of landmarks such that for all $u\in V$, $|B_L(u)|,|C_L(u)| = \tilde{O}(n^{1/4})$. Such a set can be obtained with $|L|=\tilde{O}(n^{3/4})$, by choosing in Lemma 1 $s=(n\log n)^{3/4}$ and $W=V$.

Let $E(u)=E_V(u,2n^{2/4}\log n)$ and let $e(u)=\max\{d(u,v) : v\in E(u)\}$ be the radius of the ball $E(u)$. Let $\ell(u)$ be the closest vertex in L to u (this vertex defines the radius of $B_L(u)$). Let $c : V\to[1,n^{1/4}]$ be a hash function that maps vertices into $n^{1/4}$ colors, with the following two properties:

(1) $\forall u\in V$ and $j\in[1,n^{1/4}]$, $E(u)$ contains a vertex $\ell\in L$ such that $c(\ell)=j$;
(2) $\forall j\in[1,n^{1/4}]$, the number of vertices with $c(u)=j$ is at most $2n^{3/4}$.

Clearly a $O(\log n)$-wise independent hash function has this property but known constructions of such functions require non-constant time (poly-log) to evaluate $c(u)$. To get constant time, we use the construction of [1] to obtain a hash function that can be represented using $O(n^{3/4})$ bits and allows computing $c(u)$ is constant time.

For every $x\in V$, let $T(x)$ be a spanning shortest path tree rooted at x.

High Level Idea. Given a source u and target v there will be three cases. First, if $E(u)\cap B(v)\neq\varnothing$ then we will route with affine stretch d (shortest path). Otherwise the balls $E(u)$ and $B(v)$ are disjoint. If the radius of $B(v)$ is smaller than the one of $E(u)$ then we route on shortest path to the closest landmark to v and then on a shortest path to v, so the affine stretch is $2d+1$. Otherwise, to get affine stretch $2d+1$ we need to route on shortest path to some landmark $w\in E(u)$ and then on shortest path from w to v. The problem is that w needs to know the label of v on $T(w)$. The label of v cannot store this information because there are too many landmarks. So this information must be stored in w. So the landmarks in $E(u)$ need to collectively store labels of all n destinations. By using a random hash function c we speared this $\tilde{O}(n)$ bits of information over the $O(n^{1/4})$ landmarks in $E(u)$ so that each landmark in $E(u)$ stores only $\tilde{O}(n^{3/4})$ bits of information.

Label for a vertex v is $\langle v, \ell(v), d(v, \ell(v)), \lambda(T(\ell(v)), v)\rangle$ of $O(\log^2 n/\log\log n)$ bits from Lemma 2.

Storage for a Vertex u:

(1) For every $x \in E(u)$, store routing information $\mu(T(x), u)$ of the tree $T(x)$. Also store $e(u)$.
(2) For every $v \in C_L(w)$ such that $w \in E(u)$ and w is on the shortest path from u to v, store $\langle v, w, \lambda(T(w), v)\rangle$.
(3) For every vertex $\ell \in L$, store routing information $\mu(T(\ell), u)$ of the tree $T(\ell)$.
(4) If $u \in L$, then for every vertex v such that $c(v) = c(u)$, store $\lambda(T(u), v)$.

Storing (1) requires $\tilde{O}(n^{2/4})$ space since $|E(u)| = \tilde{O}(n^{2/4})$. (2) requires $\tilde{O}(n^{3/4})$ space since $|C_L(w)| = \tilde{O}(n^{1/4})$ for each $w \in E(u)$. (3) requires $\tilde{O}(n^{3/4})$ space since $|L| = \tilde{O}(n^{3/4})$. (4) requires $\tilde{O}(n^{3/4})$ space from the second property of c. Overall, the storage for u is $\tilde{O}(n^{3/4})$.

Routing from u to v. Given the label of v, routing from u to v at distance d is done in the following manner:

1. If exists $w \in E(u)$ such that $w \in B_L(v)$ (this can be checked using (2) and in constant time using a static dictionary), then route to w using (1) and from w to v using (2). The affine stretch is d. Otherwise, it must be that $E(u)$ and $B_L(v)$ are disjoint (here we use $x \in B_L(v)$ implies $y \in B_L(v)$ for all $d(v, y) \leqslant d(v, x)$ - note that this is not necessary true for $E(u)$). Since the graph is unweighted then $\min\{e(u), d(v, \ell(v))\} \leqslant \lceil d/2 \rceil$.
2. If $e(u) > d(v, \ell(v))$ (checked using (1) and v's label) then route to $\ell(v)$ using (3). Then route from $\ell(v)$ to v using (3) and the label of v (that contains $\lambda(T(\ell(v)), v)$). From the triangle inequality, the affine stretch is $d + 2\lceil d/2 \rceil$.
3. Otherwise $e(s) \leqslant d(v, \ell(v))$ then route to some $\ell \in L \cap E(u)$ such that $c(\ell) = c(v)$ (using (3)) then using (4) on ℓ route to v. From the triangle inequality, the affine stretch is $d + 2\lceil d/2 \rceil$.
   Note that such an ℓ exists from the first property of c.

In all the cases, the affine stretch is $d + 2\lceil d/2 \rceil \leqslant 2d + 1$ as required. Using standard dictionary and hashing techniques, each of the routing decisions above can be done in constant time.

Proof of Theorem 2 for $k = 3$. Let L_1 be a set of landmarks such that for all $u \in V$, $|B_{L_1}(u)|, |C_{L_1}(u)| = \tilde{O}(n^{1/7})$. Such a set can be obtained with $|L_1| = \tilde{O}(n^{6/7})$, by choosing in Lemma 1 $s = (n \log n)^{6/7}$ and $W = V$.

Using Lemma 1 with $W = L_1$ and $s = (n \log n)^{3/7}$ let $L_2 \subset L_1$ be a set of *super landmarks* such that: (1) $|L_2| = \tilde{O}(n^{3/7})$; (2) for all $u \in V$, $|B_{L_1,L_2}(u)| = \tilde{O}(n^{3/7})$; (3) for all $\ell \in L_1$, $|C_{L_1,L_2}(\ell)| = O(n^{4/7})$.

Let $\ell_1(u)$ be the closest node in L_1 to u. Let $\ell_2(u)$ be the closest node in L_2 to u. Let $E_1(u) = E_V(u, 2n^{2/7}\log n)$ and $e_1(u) = \max\{d(u,v) : v \in E_1(u)\}$. Let $E_2(u) = E_{L_1}(u, 2n^{3/7}\log n)$ and $e_2(u) = \max\{d(u,v) : v \in E_2(u)\}$. Let $c : V \to [1, n^{1/7}]$ be a random coloring that maps vertex into $n^{1/7}$ colors. We need the following properties:

(1) $\forall u \in V$ and $j \in [1, n^{1/7}]$, $E_1(u)$ contains a vertex $\ell \in L_1$ such that $c(\ell) = j$;
(2) $\forall u \in L_1$ and $j \in [1, n^{1/7}]$, $|\{w \in V \cap C_{L_1,L_2}(u) : c(w) = j\}| \leqslant 2n^{3/7}\log n$.

Again we use a similar construction to that of [1] to obtain c with the above properties that requires $\tilde{O}(n^{3/7})$ bits and can be evaluated in constant time.

High Level Idea. Given a source u and target v there will again be three main cases. First, if $E_1(u) \cap B_{L_1}(v) \neq \varnothing$ then we will route with affine stretch d (shortest path). Otherwise the balls $E_1(u)$ and $B_{L_1}(v)$ are disjoint. If $B_{L_1}(v)$ is smaller than $E_1(u)$ then we try to route using $\ell_1(v)$ and get affine stretch of $2d+1$, but if u does not know about $\ell_1(v)$ then we use $\ell_2(v)$. As in [16] each time we move from $\ell_i(v)$ to $\ell_{i+1}(v)$ we apply the triangle inequality an loose $4d$ each step. For $k = 3$ we do this once to obtain affine stretch of $6d+1$.

Otherwise, (as in the $k = 2$ case) we route to the $\ell \in L_1 \cap E_1(u)$ such that $c(\ell) = c(v)$. If ℓ knows about v then get affine stretch of $2d+1$. Otherwise we route back to u (and pay $2d+1$). We show that $d(v, \ell_2(v)) \leqslant d + (d+1)/2$. So we try to route using $\ell_2(v)$ to get affine stretch of $(2d+1)+(4d+1) \leqslant 6d+1$. In the general case, if u does not know about $\ell_i(v)$, then we test if u knows about $\ell_{i+1}(v)$. As in [16] each time we move from $\ell_i(v)$ to $\ell_{i+1}(v)$ we apply the triangle inequality and loose $4d$ in each such iteration.

Label for a vertex v is $\langle v, \ell_1(v), d(v,\ell_1(v)), \lambda(T(\ell_1(v)), v), \ell_2(v), \lambda(T(\ell_2(v)), v)\rangle$, which is $O(\log^2 n / \log\log n)$ bits from Lemma 2.

Storage for a Vertex u:

(1) For every vertex $x \in E_1(u)$, store routing information $\mu(T(x), u)$ of the tree $T(x)$. Also store $e_1(u)$.
(2) For every $v \in C_{L_1}(w)$ such that $w \in E_1(u)$ and w is on the shortest path from u to v, store $\langle v, w, \lambda(T(w), v)\rangle$. This is $\tilde{O}(n^{3/7})$ storage since $|E_1(u)| \cdot \max|C_{L_1}(w)| = \tilde{O}(n^{3/7})$.
(3) For every vertex $\ell \in E_2(u)$, store routing information $\mu(T(\ell), u)$ of the tree $T(\ell)$. This is $\tilde{O}(n^{3/7})$ storage since $|E_2(u)| = \tilde{O}(n^{3/7})$.
(4) If $u \in L_1$ then for every vertex $v \in C_{L_1,L_2}(u)$ such that $c(v) = c(u)$ store $\lambda(T(u), v)$. This is $\tilde{O}(n^{3/7})$ storage by the property of c.
(5) For every vertex $\ell \in L_2$, store routing information $\mu(T(\ell), u)$ of the tree $T(\ell)$. This is $\tilde{O}(n^{3/7})$ storage since $|L_2| = \tilde{O}(n^{3/7})$.

Routing from u to v. Given the label of v, routing from u to v at distance d is done in the following manner:

1. If exists $w \in E_1(u)$ such that $w \in B_{L_1}(v)$ then route to w using (1) and from w to v using (2). The routing has affine stretch of d (shortest path). Otherwise it must be that $E(u)$ and $B_L(v)$ are disjoint. Since the graph is unweighted then $\min\{e(u), d(v, \ell(v))\} \leqslant \lceil d/2 \rceil$.
2. If $e_1(u) > d(v, \ell_1(v))$ (this can be checked using (1) and v's label) then:
   (a) If $\ell_1(v) \in E_2(u)$ then route to $\ell_1(v)$ using (3) and from $\ell_1(v)$ to v using (3) and the label of v. From the triangle inequality, the affine stretch is $d + 2\lceil d/2 \rceil$.
   (b) Otherwise from the triangle inequity it must be that $e_2(u) \leqslant d + \lceil d/2 \rceil$ and since $|E_2(u)| = \Theta(n^{3/7})$ it follows that $d(u, L_2) \leqslant e_2(u)$. So from the triangle inequality $d(v, L_2) \leqslant 2d + \lceil d/2 \rceil$. So using (5) route from u to $\ell_2(v)$ and from $\ell_2(v)$ to v using (5) and using v's label that contains $\lambda(T(\ell_2(v)), v))$. From the triangle inequality, the affine stretch is at most $(3d + \lceil d/2 \rceil) + (2d + \lceil d/2 \rceil) \leqslant 5d + 2\lceil d/2 \rceil \leqslant 6d + 1$ as required.
3. Otherwise $e_1(u) \leqslant d(v, \ell_1(v))$ then:
   (a) Route to some $\ell \in L_1 \cap E_1(u)$ such that $c(\ell) = c(v)$ (using (3)). Note that such a ℓ must exist from the properties of c. If $v \in C_{L_1,L_2}(\ell)$ then using (4), route on $T(\ell)$ to v. From the triangle inequality, the affine stretch is $d + 2\lceil d/2 \rceil$.
   (b) Otherwise, $\ell \notin B_{L_1,L_2}(v)$ hence from the triangle inequality, $d(v, \ell_2(v)) \leqslant 2d + \lceil d/2 \rceil$. So route from ℓ back to u on $T(\ell)$ using (3) and u's label. From u route to $\ell_2(v)$ and from $\ell_2(v)$ to v using (5) and since v's label contains $\lambda(T(\ell_2(v)), v))$. The affine stretch of going to ℓ and back to u is at most $d + \lceil d/2 \rceil$. The affine stretch of going from u to $\ell_2(v)$ and then to v is at most $(2d + \lceil d/2 \rceil) + (d + \lceil d/2 \rceil) \leqslant 3d + 2\lceil d/2 \rceil$. So the total affine stretch is $4d + 3\lceil d/2 \rceil \leqslant 5d + 1 + \lceil d/2 \rceil \leqslant 5d + 2\lceil d/2 \rceil \leqslant 6d + 1$.

In all the cases, the affine stretch is $5d+2\lceil d/2 \rceil \leqslant 6d+1$ as required. Again using standard dictionary and hashing techniques all routing decision can be made in constant time.

4 Conclusion

We have provided new distance label and compact routing schemes that provide affine stretch guarantees for unweighted graphs. Our results obtain new space-stretch trade-offs that neither dominate or are dominated by the best known previous constructions. So there exist restrictions on either the space or the stretch, for which our results provide improved bounds.

There are several questions that remain open. We believe the most intriguing problem we leave open is:

> Design a constant time (or poly-log time) approximate distance oracle for unweighted graphs with affine stretch $3d + 2$ and space $o(n^{3/2})$, or show that such construction is not possible.

Note that it is easy to construct a spanner with $O(n^{4/3})$ edges and affine stretch $3d+2$, for instance using the construction of [5] for $k=3$ with size $O(n^{1+1/k})$ and stretch $kd+k-1$.

When focusing just on affine stretch of $\alpha d+1$, the natural question is whether the memory requirements of our schemes can be improved. Specifically the current state of upper bounds indicates four different bounds: for spanners, distance oracles, fast query labels, and routing schemes.

For this abstract we did not optimize the time construction and poly-log factors in the label or routing table size. We also plan to extend our result to weighted sparse graphs using the same approach as [14].

References

1. Abraham, I., Gavoille, C., Malkhi, D., Nisan, N., Thorup, M.: Compact name-independent routing with minimum stretch. ACM Trans. on Algo. 3 (2008)
2. Aingworth, D., Chekuri, C., Indyk, P., Motwani, R.: Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM J. on Comp. 28 (1999)
3. Althöfer, I., Das, G., Dobkin, D.P., Joseph, D.A., Soares, J.: On sparse spanners of weighted graphs. Discr. & Comp. Geom. 9, 81 (1993)
4. Awerbuch, B., Peleg, D.: Sparse partitions. In: FOCS, p. 503. IEEE Press, Los Alamitos (1990)
5. Baswana, S., Kavitha, T., Mehlhorn, K., Pettie, S.: Additive spanners and (α, β)-spanners. ACM Trans. on Algo. 7, A.5 (2010)
6. Elkin, M.: Computing almost shortest paths. ACM Trans. on Algo. 1, 323 (2005)
7. Elkin, M., Peleg, D.: $(1+\epsilon, \beta)$-spanner constructions for general graphs. SIAM J. on Comp. 33, 608 (2004)
8. Erdös, P.: Extremal problems in graph theory, p. 29. Publ. House Cszechoslovak Acad. Sci., Prague (1964)
9. Fraigniaud, P., Gavoille, C.: Routing in trees. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 757–772. Springer, Heidelberg (2001)
10. Gavoille, C., Sommer, C.: Sparse spanners vs. compact routing. In: SPAA, p. 225. ACM Press, New York (2011)
11. Kleinrock, L., Kamoun, F.: Hierarchical routing for large networks; performance evaluation and optimization. Computer Networks 1, 155 (1977)
12. Mendel, M., Naor, A.: Ramsey partitions and proximity data structures. In: FOCS, p. 109. IEEE Comp. Soc. Press, Los Alamitos (2006)
13. Pettie, S.: Low distortion spanners. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 78–89. Springer, Heidelberg (2007)
14. Pătraşcu, M., Roditty, L.: Distance oracles beyond the Thorup-Zwick bound. In: FOCS, p. 815. IEEE Comp. Soc. Press, Los Alamitos (2010)
15. Sommer, C., Verbin, E., Yu, W.: Distance oracles for sparse graphs. In: FOCS, p. 703. IEEE Comp. Soc. Press, Los Alamitos (2009)
16. Thorup, M., Zwick, U.: Compact routing schemes. In: SPAA, p. 1. ACM Press, New York (2001)
17. Thorup, M., Zwick, U.: Approximate distance oracles. J. ACM 52, 1 (2005)
18. Thorup, M., Zwick, U.: Spanners and emulators with sublinear distance errors. In: SODA, p.802. ACM-SIAM (2006)
19. Woodruff, D.P.: Lower bounds for additive spanners, emulators, and more. In: FOCS, p. 389. IEEE Comp. Soc. Press, Los Alamitos (2006)

The Complexity of Data Aggregation in Directed Networks

Fabian Kuhn[1] and Rotem Oshman[2,⋆]

[1] University of Lugano, Switzerland
[2] Massachusetts Institute of Technology, USA

Abstract. We study problems of data aggregation, such as approximate counting and computing the minimum input value, in synchronous directed networks with bounded message bandwidth $B = \Omega(\log n)$. In *undirected* networks of diameter D, many such problems can easily be solved in $O(D)$ rounds, using $O(\log n)$-size messages. We show that for directed networks this is not the case: when the bandwidth B is small, several classical data aggregation problems have a time complexity that depends polynomially on the size of the network, even when the diameter of the network is constant. We show that computing an ϵ-approximation to the size n of the network requires $\Omega(\min\left\{n, 1/\epsilon^2\right\}/B)$ rounds, even in networks of diameter 2. We also show that computing a sensitive function (e.g., minimum and maximum) requires $\Omega(\sqrt{n/B})$ rounds in networks of diameter 2, provided that the diameter is *not known in advance* to be $o(\sqrt{n/B})$. Our lower bounds are established by reduction from several well-known problems in communication complexity. On the positive side, we give a nearly optimal $\tilde{O}(D + \sqrt{n/B})$-round algorithm for computing simple sensitive functions using messages of size $B = \Omega(\log N)$, where N is a loose upper bound on the size of the network and D is the diameter.

1 Introduction

Consider a wireless network comprising two base stations, transmitting at high power, and an unknown number of client devices which communicate only with the base stations. The base stations are received at all devices, and each client device is received by at least one base station. However, due to power constraints, the clients are not necessarily received at both stations. The bandwidth of each base station is limited, allowing it to send only a certain number B of bits per timeslot. How many timeslots are required for the base stations to determine the approximate number of clients? We study this problem and other data aggregation problems in *directed networks*, where communication is not necessarily bidirectional.

Data aggregation tasks are central to many distributed systems; for example, a peer-to-peer network might require information about the number of clients that have a local copy of a file, and a sensor network might need to verify that an anomalous reading was detected by a certain percentage of sensors before reporting it. With the increasing

⋆ Rotem Oshman was supported by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

availability of dynamic, large-scale distributed systems, efficient data aggregation has become a particularly interesting challenge.

Classically, data aggregation has been studied in networks with bidirectional communication links. In this setting the method of choice is to first construct a spanning tree of the network graph, and then perform distributed data aggregation "up the tree". In a synchronous undirected network, if computation is initiated by some node, a global broadcast starting at the initiating node induces a breadth-first search spanning tree of the network. Basic aggregation functions, such as the minimum, maximum, sum, or average of values distributed across the nodes of the system, can then efficiently be computed by a simple convergecast on the tree. Even when the message bandwidth is quite restricted (e.g., if only a constant number of data items can be sent in a single message), this method allows any of the functions above to be computed in $O(D)$ rounds in networks of diameter D. Network properties such as the size of the network and the diameter D itself can also be determined in $O(D)$ time using small messages. In fact, in [1] Awerbuch observes that computing certain aggregation functions and computing a spanning tree are intimately related problems, whose time and message complexities are within constant factors of each other. This makes the spanning-tree/convergecast approach a canonical solution of sorts.

The situation changes significantly when communication is not necessarily bidirectional. Constructing a rooted directed spanning tree becomes much more challenging, as it is much harder for the sender of a message to obtain feedback from the recipients, or even to determine who are the recipients. In this paper we show that in contrast to undirected networks, in directed networks with restricted bandwidth it is not always desirable to aggregate data by first computing a rooted spanning tree; for some functions, such as minimum and maximum, it is faster to compute the aggregate by other means. Moreover, we show that the time complexity of computing an aggregate with restricted bandwidth is not governed by the diameter of the network alone; for small-diameter networks, the time complexity of computing certain aggregates is dominated by a factor polynomial in n, the size of the network. We are particularly interested in the effect of *initial knowledge*, i.e., whether or not the problem becomes easier if parameters such as the size or diameter of the network are known in advance.

The paper is organized as follows. In Section 2 we discuss related work. In Section 3 we introduce the model and problems studied in the paper, and review several results in communication complexity that form the basis for our lower bounds. In Section 4 we consider the problems of exact and approximate counting, when the diameter of the network is known to be 2; we show that computing an ϵ-approximate count with constant probability requires $\Omega(\min\left\{n, 1/\epsilon^2\right\}/B)$ rounds where B is the message bandwidth. Our lower bound implies that computing a rooted spanning tree in networks of diameter 2 requires $\Omega(n/B)$ rounds.

In Section 5 we turn our attention to computing sensitive functions in networks of unknown diameter. Informally, a function is *globally sensitive* if its value depends on all the inputs, and *ϵ-sensitive* if its value depends on an ϵ-fraction of inputs. In undirected networks, or even in directed networks of known diameter D, some globally-sensitive functions can be computed in $O(D)$ time with only single-bit messages. We show that for directed networks of *unknown* diameter the picture is quite different:

$\Omega(\sqrt{n/B})$ rounds are required, even when the diameter of the network is 2 (but this fact is not known in advance). This lower bounds holds for randomized computation of any globally-sensitive function and for deterministic computation of any ϵ-sensitive function where $\epsilon \in (0, 1/2)$. The lower bound holds even when the size n of the network is known in advance and the UID space is $1, \ldots, n$.

Finally, in Section 5.2 we give a randomized algorithm for the problem of determining when a node has been causally influenced by all nodes in the graph. This condition is necessary to compute a globally-sensitive function, and sufficient to compute simple functions such as minimum or maximum. The algorithm requires $D + \tilde{O}(\sqrt{n/B})$ rounds w.h.p., nearly matching our lower bound. For lack of space, some of the proofs are omitted here, and appear in the full version of this paper.

2 Background and Related Work

Distributed data aggregation and spanning tree computation. Early work on these problems was concerned with their *message complexity*, that is, the total number of messages sent by all processes, as well as their time complexity. Awerbuch observed in [1] that in undirected networks, the message and time complexity of leader election, computing a distributive sensitive function (e.g., minimum or maximum) and counting are all within a constant factor of the complexity of finding a spanning tree in the network. It is also shown in, e.g., [1,3] that the time complexity of these problems in undirected networks is $\Theta(n)$ and the message complexity is $\Theta(m + n \log n)$ in networks of size n with m edges. However, the $\Omega(n)$ lower bound is obtained in networks of diameter $\Omega(n)$, and the message complexity lower bound does not yield a non-trivial bound in our model. In a synchronous undirected network of diameter D edges, it is possible to construct a breadth-first search spanning tree in $O(D)$ rounds, even if the diameter and size of the network are not known in advance. Using such a tree, functions such as minimum, maximum, sum, or average can all be computed in time $O(D)$. Based on a pre-computed spanning tree, researchers have also considered the computation of more complicated functions such as the median or the mode [7,8,12,13,15,16,17].

Communication complexity. A two-player communication game involves two players, Alice and Bob, which are given private inputs x, y and must compute some joint function of their inputs, $f(x, y)$. In order to compute f the players communicate over several rounds, and are charged for the total number of bits exchanged. The *deterministic communication complexity* of f is the worst-case number of bits exchanged in any deterministic protocol for computing f. The *randomized communication complexity* is defined similarly; in the current paper we are interested in randomized algorithms that err with constant probability.

Communication complexity lower bounds have often been used to obtain lower bounds in distributed computing. The classical reduction technique (see, e.g., [10]) partitions the network into two parts, with each player simulating the nodes on one side of the cut. The input to each player is reflected in the structure of its part of the network or in the input to the network nodes it simulates, and the output or behavior of the distributed algorithm is used , and the communication-complexity lower bound then shows that a certain amount of information must cross the cut. For example, this technique is used in [13] to obtain a lower bound on the complexity of computing the number of distinct elements in the input.

The reductions we give here are quite different in nature. Instead of partitioning the network, the players simulate non-disjoint sets of nodes. Care must be taken to ensure that information about one player's private input does not "leak" to the other player through nodes that both players simulate; this aspect of our reductions strongly relies on the fact that the network is directed.

3 Preliminaries

Network model. We model a synchronous directed network as a strongly connected directed graph $G = (V, E)$, where $E \subseteq V^2$. We use $N^d(v) = \{u \in V \mid \text{dist}(u, v) \leq d\}$ to denote the *d-in-neighborhood* of v, that is, the set of nodes whose distance to v is at most d. Nodes communicate by local broadcast: in each round, every node u sends a single message of size at most B, where $B = \Omega(\log n)$, and this message is delivered to all nodes v such that $(u, v) \in E$. (Each node does not know which nodes receive its message, i.e., it does not know its set of out-neighbors.) We assume that nodes and communication links are reliable and do not fail during an execution.

In the sequel we often refer to algorithms whose correctness is only guaranteed in networks that satisfy some fixed bound on the size or diameter of the network. In this case we say that the bound is *known a priori* (or *known in advance*). Our lower bounds assume that each node has a unique identifier (UID) drawn from some UID space $1, \ldots, N$, where N is an upper bound on the size of the network that is known in advance. For convenience, we assume the existence of two distinguished UIDs $a, b \notin [N]$; our reductions "embed" the two players in the graph as nodes a and b respectively. Some of our lower bounds allow for the case where $N = n$, i.e., the exact size of the network is known to all nodes and the UID space is $1, \ldots, n$. In contrast, the algorithm in Section 5.2 requires only a loose upper bound $N \geq n$ and does not use UIDs at all.

Problem statements. We are interested in the following distributed problems.

- *ϵ-approximate counting*: nodes are initially provided with some loose upper bound N on the size n of the network, and each node v must eventually output an approximate count $\tilde{n}_v$ satisfying $|\tilde{n}_v - n| \leq \epsilon \cdot n$.
- Computing *globally-sensitive* functions of the input: a function is said to be *globally sensitive* if there exists an input assignment $\overline{x}$ such that changing any single coordinate of $\overline{x}$ yields a different function value. For example, the all-one input assignment witnesses the global sensitivity of computing a minimum.
- Computing *ϵ-sensitive* functions of the input: a function is *ϵ-sensitive* if there is an input assignment $\overline{x}$ such that changing any $\lceil \epsilon n \rceil$ coordinates of $\overline{x}$ yields a different function value. For example, the function that returns 1 iff at least 25% of the inputs are 1 is $(1/4)$-sensitive, as witnessed by the all-zero input assignment.

Communication complexity lower bounds. Our results rely on several celebrated lower bounds in communication complexity. Perhaps the best known lower bound concerns the Set Disjointness problem, DISJ_n, in which the players are given sets $X, Y \subseteq [n]$ (respectively) and must determine whether $X \cap Y = \emptyset$.

Theorem 1 **([5,14]).** *The randomized communication complexity of* DISJ_n *is* $\Omega(n)$.

We are also interested in a relaxed variant called Gap Set Disjointness, $\text{GAP-DISJ}_{n,g}$: here the players are given sets $X, Y \subseteq [n]$, with the *promise* that either $X \cap Y = \emptyset$ or $|X \cap Y| \geq g$. The players must determine which of these cases holds. When the gap g is large with respect to n, $\text{GAP-DISJ}_{n,g}$ is quite easy for randomized algorithms (one can use random sampling to find an element of the intersection if it is large). However, for deterministic protocols the problem remains hard even with a linear gap. (This fact appears to be folklore in the communication complexity community; we include a proof in the full version of this paper.)

Theorem 2. *For any constant* $\epsilon \in (0, 1/2)$*, the deterministic communication complexity of* $\text{GAP-DISJ}_{n,(1/2-\epsilon)n}$ *is* $\Omega(n)$.

The final problem is GAP-HAMMING-DISTANCE, denoted $\text{GHD}_{n,g}$, where the players receive vectors $x, y \in \{0, 1\}^n$ and must determine whether the Hamming distance $\Delta(x, y)$ satisfies $\Delta(x, y) > n/2 + g$ or whether $\Delta(x, y) \leq n/2 - g$. (If neither holds, any answer is allowed.) Characterizing the randomized communication complexity of GHD remained an open problem for a long time after its introduction in [4] (for the case $g = \sqrt{n}$, which is in some sense the most interesting setting), until in [2], Chakrabarti and Regev proved the following lower bound.

Theorem 3 **([2]).** *For any* $g \leq n$*, the randomized communication complexity of* $\text{GHD}_{n,g}$ *is* $\Omega(\min\{n, n^2/g^2\})$.

The reductions in this paper are *public-coin protocols*: they assume that Alice and Bob have access to a shared random string (of unbounded length). The lower bounds above are stated for *private-coin protocols*, where each player has its own private randomness. However, any public-coin protocol can be transformed into a private-coin protocol at the cost of $O(\log n)$ additional bits [10], so the distinction is mostly immaterial for our purposes.

4 Approximate and Exact Counting

We begin by describing a lower bound for ϵ-approximate counting or exact counting. In this setting we assume that nodes know some loose upper bound $N \geq n$ on the size of the network, and must determine the exact or approximate size. Since exact counting is a special case of approximate counting, we describe the lower bound for approximate counting, and later discuss exact counting.

The lower bound is obtained by reduction from $\text{GHD}_{N,\epsilon N}$. Suppose we are given an ϵ-approximate counting algorithm $\mathcal{A}$. Given an instance (x, y) of $\text{GHD}_{N,\epsilon N}$, we construct a network $G_{x,y}$, in which Alice and Bob jointly simulate the execution of $\mathcal{A}$. When $\mathcal{A}$ terminates, Alice and Bob use the output of $\mathcal{A}$ to determine the correct answer to GHD on the instance (x, y). Since Alice knows only her input x and Bob knows only y, neither player knows the complete topology of the network $G_{x,y}$, which depends on both x and y. The players therefore cooperate to simulate the execution of $\mathcal{A}$ in $G_{x,y}$.

Let $X, Y \subseteq [N]$ be the sets whose characteristic vectors are x and y, respectively. The network $G_{x,y}$ is given by $G_{x,y} = (V_{x,y}, E_{x,y})$, where $V_{x,y} = X \cup Y \cup \{a, b\}$ (for

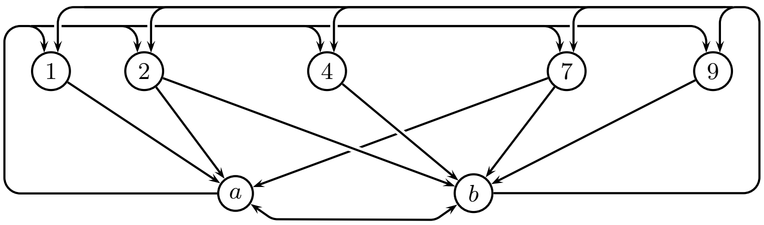


Fig. 1. The network $G_{x,y}$ for $x = 110000100, y = 010100101$ (i.e., $X = \{1,2,7\}, Y = \{2,4,7,9\}$)

$a, b \notin [N]$), and $E_{x,y} = (\{a\} \times V_{x,y}) \cup (\{b\} \times V_{x,y}) \cup (X \times \{a\}) \cup (Y \times \{b\})$ (see Fig. 1).

The Hamming distance $\Delta(x,y)$ is closely related to the size of $G_{x,y}$:

Lemma 1. *For all $(x,y) \in (\{0,1\}^N)^2$, the graph $G_{x,y}$ is strongly connected, its diameter is 2, and its size is $|V_{x,y}| = (\|x\|_1 + \|y\|_1 + \Delta(x,y))/2 + 2$.*

Next we show that an efficient algorithm for approximating the size of diameter 2 networks leads to an efficient protocol for $\mathrm{GHD}_{N,\epsilon N}$.

Lemma 2. *Given an ϵ-approximate counting algorithm $\mathcal{A}$ which outputs a correct answer after t rounds with probability at least $1 - \delta$, one can construct a public-coin protocol for $\mathrm{GHD}_{N,\epsilon N}$ which exchanges a total of $O(Bt + \log N)$ bits and succeeds with probability $1 - \delta$.*

Proof. Given an instance (x,y), Alice and Bob simulate the execution of $\mathcal{A}$ in $G_{x,y}$ as follows. Alice locally simulates the nodes in $X \cup \{a\}$, and Bob locally simulates the nodes in $Y \cup \{b\}$. The shared random string is used to provide the randomness of all nodes in the network. (Since Alice and Bob do not initially know which of the nodes $\{1,\ldots,N\}$ are present, we interpret the shared random string as containing the randomness of each node $1,\ldots,N$ regardless of whether or not the node is in $X \cup Y$.) Notice that there can be some overlap, $X \cap Y$, which is simulated by both players independently.

The initial states of all nodes in $X \cup \{a\}$ and in $Y \cup \{b\}$ are known to Alice and Bob, respectively, because they depend only on the UIDs of these nodes and on the shared randomness. Each round of $\mathcal{A}$ is simulated as follows:

- Based on the states of their local simulations, Alice and Bob compute the messages sent by the nodes in $X \cup \{a\}$ and in $Y \cup \{b\}$, respectively.
- Alice sends to Bob the message sent by node a, and Bob sends to Alice the message sent by b. Following this exchange, Alice and Bob have all the messages received by each node they need to simulate.
- The players update the states of their local simulations by feeding to each node the messages it receives in $G_{x,y}$: the nodes of $X \cup Y$ receive the messages sent by a and b; node a receives the messages sent by nodes in $X \cup \{b\}$; and node b receives the messages sent by nodes in $Y \cup \{a\}$. (Note that Alice knows X and Bob knows

Y, so the two players know which messages are supposed to be received by nodes a, b, respectively.)

Although Alice and Bob do not directly exchange information about the states of nodes in $X \cap Y$ — indeed, they do not *know* which nodes are in $X \cap Y$, and this is what makes the problem difficult — still their local simulations agree on the states of these nodes.

With probability at least $1 - \delta$, after t rounds of the simulation node a halts and outputs an approximate count $\tilde{n}$ which satisfies $|\tilde{n} - n| \leq \epsilon n$. When node a halts, Alice sends $\tilde{n}$ to Bob, and in addition Alice and Bob send each other $|X| = \|x\|_1$ and $|Y| = \|y\|_1$ (respectively). Let $\tilde{\Delta} = 2(\tilde{n} - 2) - \|x\|_1 - \|y\|_1$. Both players output 0 if $\tilde{\Delta} < N/2$, and 1 if $\tilde{\Delta} \geq N/2$. (If node a fails to halt after t rounds, the players output an arbitrary answer.)

If $|\tilde{n} - n| \leq \epsilon n$ then Lemma 1 shows that $|\tilde{\Delta} - \Delta(x, y)| = 2|\tilde{n} - n| \leq 2\epsilon n \leq 2\epsilon N$. Hence, with probability at least $1 - \delta$, the players output the correct answer: if $\Delta(x, y) \geq N/2 + 2\epsilon N$ then $\tilde{\Delta} \geq N/2$, and if $\Delta(x, y) < N/2 - 2\epsilon N$ then $\tilde{\Delta} < N/2$.

The total number of bits sent during the protocol is $2Bt + 2\log(N)$. In addition, to transform the protocol into a private-coin protocol we require $O(\log N)$ additional bits. The communication complexity is therefore $O(Bt + \log N)$. □

Although our reduction is stated in terms of the upper bound N (we reduce from $\text{GHD}_{N,\epsilon N}$), the "hard" instances are the ones where n is roughly linear in N; it is always possible to solve GHD by exchanging the coordinates of indices i such that $x_i = 1$ or $y_i = 1$, and hence when $|X \cup Y| = n$ the problem can easily be solved in $O(n \log N)$ bits. It is therefore more informative to state our lower bound in terms of the actual size n of the network. From Theorem 3 and the reduction above, we obtain the following lower bound.

Theorem 4. *If $B = \Omega(\log N)$, a randomized algorithm for computing an ϵ-approximate count requires $\Omega((\min\{n, 1/\epsilon^2\}/B)$ rounds to succeed with probability 2/3 in networks of diameter 2.*

Remarks. The *deterministic* communication complexity of $\text{GHD}_{N,g}$ is $\Omega(N)$ even when $g = c \cdot N$ for a sufficiently small constant c [2]; therefore deterministically computing an ϵ-approximate count for ϵ a sufficiently small constant requires $\Omega(n/B)$ rounds. As for *exact* counting (deterministic or randomized), computing the exact count is as hard as computing a $(1/n)$-approximate count, so $\Omega(n/B)$ rounds are required.

The lower bound of Theorem 4 is nearly tight if the diameter of the network is known. An algorithm for ϵ-approximate counting is given in [11]; the algorithm of [11] sends messages containing real numbers, but using a rounding scheme to bound the size of messages (see [9]), one obtains an $\tilde{O}(D + \min\{n, 1/\epsilon^2\}/B)$-round algorithm for networks of known diameter D. For the case where the diameter is unknown, we obtain a stronger lower bound in the next section.

Finally, the reduction from Lemma 2 also shows that finding a rooted spanning tree in directed networks is hard even when the diameter of the network is known *a priori* to be 2. In the network $G_{x,y}$, the nodes of $X \cup Y$ are not connected to each other; therefore any rooted spanning tree of $G_{x,y}$ has diameter at most 3, as each node of $X \cup Y$ except possibly the root must have either a or b as its parent in the tree. If one can find a rooted spanning tree of $G_{x,y}$ in t rounds, then an exact count can be computed in $t + 3$ rounds by finding such a tree and then "summing up the tree" (convergecast). Since exact counting requires $\Omega(n/B)$ rounds, so does computing a rooted spanning tree. In the full version of this paper we show that this lower bound continues to hold when the size of the network is known *a priori*, provided that the UID space is of size at least $(1 + \epsilon)n$ for some arbitrarily small constant ϵ.

5 Computing Sensitive Functions

In this section we study the complexity of computing sensitive functions, such as the minimum or maximum input value. In contrast to the previous section, here we are interested in instances where the diameter of the network is not known *a priori* to be small, but the algorithm is deployed in a network that *does* in practice have a small diameter. We will show that in such cases it is not possible to exploit the small diameter of the network; the worst-case running time of the algorithm must be $\Omega(D + \sqrt{n/B})$. We also give a nearly-matching algorithm for computing simple sensitive functions.

Let f be a globally-sensitive function, and let $\bar{x}$ be an input assignment under which changing any node's input changes the value of f (i.e., for all $\bar{y} \neq \bar{x}$ we have $f(\bar{x}) \neq f(\bar{y})$). In any execution where the input is $\bar{x}$, at time t, a node v can only know the value of f if $N^t(v) = V$, that is, if t rounds are sufficient for a message from any node in the network to reach node v; otherwise there is some node whose input node v cannot know at time t, and this node's input may determine the value of f. Similarly, if f is ϵ-sensitive, there exists an input assignment under which no node can know the value of f at time t unless $|N^t(v)| > (1 - \epsilon)n$. This motivates us to study the following problem:

Definition 1 (Hearing from m nodes). *In the* Hear-from-m-nodes *problem, denoted* HF$_m$*, each node v in the network must halt at some time t such that $|N^t(v)| \geq m$.*

The worst-case time complexity of computing a globally-sensitive function is at least the worst-case time complexity of solving HF$_n$, and similarly for ϵ-sensitive functions and HF$_{(1-\epsilon)n}$. (In fact, computing an ϵ-sensitive function can require hearing from *strictly more* than $(1 - \epsilon)n$ nodes.) Of course, HF$_n$ can easily be solved by having all nodes wait until time $n - 1$; however, we are interested here in efficient solutions, which terminate faster in networks with smaller diameter (recall, however, that the diameter is not known in advance).

5.1 Lower Bounds on Computing a Sensitive Function

In this section we show that even when the diameter of the network is 2, *learning* that the diameter is 2 requires $\Omega(\sqrt{n/B})$ rounds in the worst case. More formally, we show that when the size of the network is known, the UID space is $1, \ldots, n$, and no *a priori* bound on the diameter is known,

(a) Any randomized algorithm for HF$_n$ requires $\Omega(\sqrt{n/B})$ rounds to succeed with constant probability, even when executed in a network of diameter 2; and

(b) For any $\epsilon \in (0, 1/2)$, any deterministic algorithm for $\mathrm{HF}_{(1-\epsilon)n}$ requires $\Omega(\sqrt{n/B})$ rounds, again when executed in networks of diameter 2.

(Of course, in networks of diameter 2 we have $|N^2(v)| = n$ for all nodes v, so $t = 2$ is sufficient; however, this fact is not known to the algorithm in advance.)

Fix an algorithm $\mathcal{A}$ for HF_m and a network size $n \geq m$. We describe a reduction from Set Disjointness or Gap Set Disjointness, which we will use to show both the hardness of HF_n for randomized algorithms and the hardness of $\mathrm{HF}_{(1-\epsilon)n}$ for deterministic algorithms.

As in Section 4, in the reduction we construct a network G based on the instance of Set Disjointness given to Alice and Bob. The two players then simulate the execution of $\mathcal{A}$ in G, and output an answer to Set Disjointness (or Gap Set Disjointness) based on the behavior of $\mathcal{A}$ in G — in this case, based on the time when $\mathcal{A}$ terminates. We now describe the construction of the network and the simulation used by Alice and Bob.

The construction has several parameters. First, let $t_{\mathcal{A}}$ be the number of rounds such that when $\mathcal{A}$ is executed in a network of size n with node UIDs $1, \ldots, n, a, b$ (as before we add UIDs a, b for convenience), with probability at least $2/3$ all nodes halt by time $t_{\mathcal{A}}$. Based on $t_{\mathcal{A}}$ and on m, we choose a *segment length* $s \geq t_{\mathcal{A}} + 1$ which will be fixed later. Informally, in the reduction nodes must distinguish diameter 2 networks from diameter $s + 2$, and we will show that this requires $\Omega(n/s)$ rounds in the worst-case.

Assume for simplicity that s divides n. We divide the nodes $1, \ldots, n$ into *segments* $S_1, \ldots, S_{n/s}$, each of size s, where $S_i := \{(i-1) \cdot s + 1, (i-1) \cdot s + 2, \ldots, i \cdot s\}$. Each segment S_i is further subdivided into two parts: a *back end* S_i^B containing nodes $(i-1) \cdots + 1, \ldots, i \cdot s - t_{\mathcal{A}}$, and a *front-end* S_i^F containing the remaining nodes, $i \cdot s - t_{\mathcal{A}} + 1, \ldots, i \cdot s$. In the sequel we implicitly use wrap-around (i.e., $\bmod n$ arithmetic) for node indices, so that $-1 \equiv n$, $-2 \equiv n - 1$, and so on.

We are now ready to describe the reduction itself. The reduction is from $\mathrm{D\textsc{isj}}_{n/s}$, that is, Set Disjointness (or Gap Set Disjointness) with a universe of n/s elements; each segment S_i represents a single element of the universe. Given an instance (x, y) of $\mathrm{D\textsc{isj}}_{n/s}$, we define a network $G_{s,x,y} := (\{1, \ldots, n, a, b\}, E_{s,x,y})$ (see Fig. 2), where

– Nodes a, b have edges to all nodes of the graph.
– Nodes $1, \ldots, n$ are connected in a directed cycle: for each $i \in [n]$ we have $(i, i + 1) \in E_{s,x,y}$.
– In each segment S_i, the last node (node $i \cdot s$) is connected to node a. (This is to ensure strong connectivity and a bound of $s + 2$ on the diameter.)
– For all $i \notin X$ and for all $v \in S_i$ we have $(v, a) \in E_{s,x,y}$; similarly, for all $i \notin Y$ and for all $v \in S_i$ we have $(v, b) \in E_{s,x,y}$.

Here, X and Y are the sets whose characteristic vectors are x, y respectively.

With the exception of the last node in each segment (which is always connected to node a), the nodes in segment S_i are connected to node a iff Alice did *not* receive i in her input, and connected to node b iff Bob did not receive i in his input. Therefore, if there exists an element i in the intersection $X \cap Y = \overline{\overline{X} \cup \overline{Y}}$, the nodes of the corresponding segment S_i, with the exception of the last node, will not be connected to either node a or node b. These nodes are only connected to the rest of the graph by the cycle edges $(i-1) \cdot s + 1 \rightarrow (i-1) \cdot s + 2 \rightarrow \ldots \rightarrow i \cdot s$. Consequently the diameter

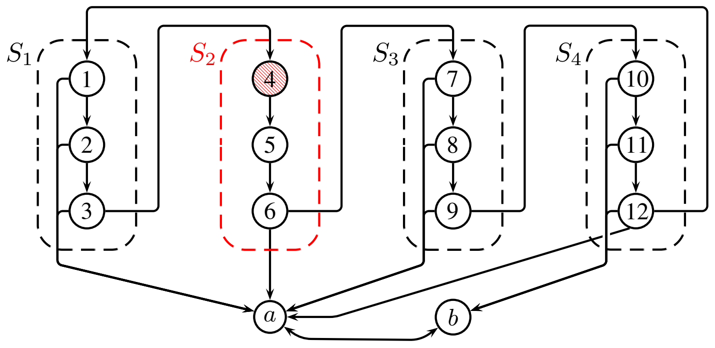


Fig. 2. The network $G_{s,x,y}$ from Thm. 5, with $n = 12, t_{\mathcal{A}} = 2, s = t_{\mathcal{A}} + 1 = 3$. Edges from a, b to nodes $1, \ldots, 12$ are omitted for clarity. The $\textsc{Disj}_4$ instance shown here is $X = \{2, 4\}, Y = \{1, 2, 3\}$. Since $2 \in X \cap Y$, all S_2 nodes except the last (node 6) are not connected to a or to b. Therefore $4 \notin N^{t_{\mathcal{A}}}(a)$, i.e., two rounds are not sufficient for node a to hear from node 4.

of the graph is $s + 2 > t_{\mathcal{A}}$ in this case. In $t_{\mathcal{A}}$ rounds, nodes a and b can only hear from the last $t_{\mathcal{A}}$ nodes of segment S_i, i.e., only from the front-end S_i^F; for each segment S_i such that $i \in X \cap Y$, $|S_i^B| = s - t_{\mathcal{A}}$ nodes are missing from $N^{t_{\mathcal{A}}}(a)$.

On the other hand, if $X \cap Y = \emptyset$ (or equivalently, $\overline{X} \cup \overline{Y} = \{1, \ldots, n/s\}$), all nodes in all segments are connected to either node a or node b, and the diameter of the graph is 2.

Lemma 3. *For any $x, y \in \{0, 1\}^n$,*
(a) The graph $G_{s,x,y}$ is strongly connected,
(b) For all $i \in X \cap Y$ and for all $v \in S_i^B$ we have $v \notin N^{t_{\mathcal{A}}}(a)$ and $v \notin N^{t_{\mathcal{A}}}(b)$,
(c) If $X \cap Y = \emptyset$, the diameter of $G_{s,x,y}$ is 2, and
(d) $|N^{t_{\mathcal{A}}}(a)| \le n - |X \cap Y| \cdot (s - t_{\mathcal{A}})$ (and similarly for b).

Alice and Bob simulate the execution of $\mathcal{A}$ in $G_{s,x,y}$ in a slightly different manner than in Lemma 2; here both players simulate nodes $1, \ldots, n$ regardless of the input instance, and in addition Alice simulates node a and Bob simulates node b. The remainder of the simulation is the same as in Lemma 2, and we omit the details here.

Proposition 1. *Given inputs x and y respectively, and a shared string representing the randomness of all nodes, Alice and Bob can each simulate nodes $\{a, 1, \ldots, n\}$ and $\{b, 1, \ldots, n\}$ (respectively) throughout rounds $1, \ldots, t_{\mathcal{A}}$ of the execution of $\mathcal{A}$ in $G_{s,x,y}$.*

It remains only to put the pieces together to obtain the following lower bounds.

Theorem 5. *If the diameter of the network is not known initially, any randomized algorithm for computing a globally-sensitive function requires $\Omega(\sqrt{n/B})$ rounds with probability at least $2/3$ when executed in networks of diameter 2.*

Proof. As explained above, it is sufficient to show the corresponding bound for HF_n.

Fix an algorithm $\mathcal{A}$, and let $t_{\mathcal{A}}$ be defined as above. Fix a segment length of $s := t_{\mathcal{A}} + 1$ (so that the back-end of each segment contains exactly one node).

Given an instance (x, y) of $\textsc{Disj}_{n/s}$, Alice and Bob jointly simulate the first $t_{\mathcal{A}}$ rounds in the execution of $\mathcal{A}$ in $G_{s,x,y}$ as in Proposition 1. After $t_{\mathcal{A}}$ rounds, Alice informs Bob whether or not node a has halted in the simulation. If node a has halted, the players output "$X \cap Y = \emptyset$"; otherwise they output "$X \cap Y \neq \emptyset$".

As we saw in Lemma 3, if $X \cap Y = \emptyset$ then the diameter of $G_{t_{\mathcal{A}}+1,x,y}$ is 2, so with probability at least $2/3$ all nodes halt after $t_{\mathcal{A}}$ rounds and Alice and Bob output "$X \cap Y = \emptyset$". On the other hand, if $X \cap Y \neq \emptyset$, then by time $t_{\mathcal{A}}$ node a has not heard from all nodes, as Lemma 3 shows that at least $(s - t_{\mathcal{A}}) \cdot |X \cap Y| = |X \cap Y| > 0$ nodes are missing from $N^{t_{\mathcal{A}}}(a)$. Consequently, with probability at least $2/3$, node a does not halt by time $t_{\mathcal{A}}$ and the players output "$X \cap Y \neq \emptyset$".

The total number of bits exchanged by the players in the protocol above is $2B \cdot t_{\mathcal{A}} + 1$, because Alice and Bob only send each other the messages output by nodes a and b, plus one bit needed for Alice to inform Bob whether node a has halted. An additional $O(\log(n/t_{\mathcal{A}}))$ bits are required to obtain a private-coin protocol. Since the randomized communication complexity of $\textsc{Disj}_{\lfloor n/(t_{\mathcal{A}}+1)\rfloor}$ is $\Omega(n/t_{\mathcal{A}})$, we must have $2B \cdot t_{\mathcal{A}} + 1 = \Omega(n/t_{\mathcal{A}})$, or in other words, $t_{\mathcal{A}} = \Omega(\sqrt{n/B})$. □

Theorem 6. *If the diameter of the network is initially unknown, any deterministic algorithm for computing an ϵ-sensitive function, where $\epsilon \in (0, 1/2)$ is constant, requires $\Omega(\sqrt{n/B})$ rounds when executed in networks of diameter 2.*

Proof (sketch). We prove that $\Omega(\sqrt{n/B})$ rounds are required to solve $\mathrm{HF}_{(1-\epsilon)n}$ deterministically for any $\epsilon \in (0, 1/2)$, even in networks of diameter 2. The proof is similar to that of Thm. 5, except that we now reduce from $\textsc{Gap-Disj}_{\lfloor n/s\rfloor,\epsilon'\lfloor n/s\rfloor}$ for an appropriately chosen constant $\epsilon' \in (0, 1/2)$, and the segment length s is also chosen differently.

Fix a deterministic algorithm $\mathcal{A}$ for $\mathrm{HF}_{(1-\epsilon)n}$, and let $t_{\mathcal{A}}$ be the maximal time at which the algorithm halts in any network of diameter 2. We must now choose a segment length $s = \Theta(t_{\mathcal{A}})$ so that the following conditions hold:

(a) If $X \cap Y = \emptyset$, then the diameter of $G_{s,x,y}$ is 2. This ensures that in "yes" instances, all nodes halt by time $t_{\mathcal{A}}$.

(b) If $|X \cap Y| \geq \epsilon'\lfloor n/s\rfloor$ then we have $|N^{t_{\mathcal{A}}}(a)| < (1 - \epsilon)(n + 2)$. This ensures that in "no" instances, node a *cannot* halt by time $t_{\mathcal{A}}$.

These conditions suffice for the protocol from Thm. 5 to solve $\textsc{Gap-Disj}_{\lfloor n/s\rfloor,\epsilon'\lfloor n/s\rfloor}$ as well. From Lemma 3 we see that condition (a) holds regardless of our choice of s. As for condition (b), from part (d) of Lemma 3, it is sufficient to choose $s := \alpha t_{\mathcal{A}}, \epsilon'$ so that

$$n - \epsilon' \left\lfloor \frac{n}{\alpha t_{\mathcal{A}}} \right\rfloor \cdot (\alpha - 1)t_{\mathcal{A}} < (1 - \epsilon)(n + 2).$$

There exist constants $\alpha > 1, \epsilon' \in (0, 1/2)$ satisfying this constraint (we omit the details for lack of space). For this choice of s, ϵ', the reduction from Thm. 5 yields a protocol with communication complexity $2Bt_{\mathcal{A}} + 1$ for $\textsc{Gap-Disj}_{n',\epsilon' n'}$, where $n' = \lfloor n/s\rfloor = O(n/t_{\mathcal{A}})$. Because $\textsc{Gap-Disj}$ is linearly hard for deterministic protocols even when the gap is linear in the universe size (Theorem 2), we must have $2Bt_{\mathcal{A}} + 1 = \Omega(n/t_{\mathcal{A}})$, i.e., $t_{\mathcal{A}} = \Omega(\sqrt{n/B})$. □

Remarks. The construction in this section can be modified to show a few related results.

In Theorems 5 and 6 we assumed that no upper bound on the diameter of the network is known in advance. Suppose now that some upper bound $\bar{D}$ on the diameter is known in advance. We can show that any randomized algorithm for computing a globally-sensitive function, and any deterministic algorithm for computing an ϵ-sensitive function for $\epsilon \in (0, 1/2)$, requires $\Omega(\min\left\{\bar{D}, \sqrt{n/B}\right\})$ rounds when executed in networks of diameter 2.

To see this, observe that the diameter of $G_{s,x,y}$ never exceeds $s+2$. Suppose that $\bar{D} = o(\sqrt{n/B})$ and we are given an HF$_n$-algorithm (or similarly, a deterministic HF$_{(1-\epsilon)n}$-algorithm) $\mathcal{A}$ with $t_{\mathcal{A}} < \bar{D} - 2$. If we use a segment length of $s = t_{\mathcal{A}} + 1 \leq \bar{D} - 2$, as in Thm. 5, the diameter upper bound is not violated in $G_{s,x,y}$. For this choice of s, the reduction from Thm. 5 allows us to solve $\text{Disj}_{\lfloor n/s \rfloor}$, where $\lfloor n/s \rfloor \geq \lfloor n/(\bar{D} - 2) \rfloor$, using less than $2(\bar{D} - 2)B + 1$ bits. We must have $2(\bar{D} - 2)B + 1 = \Omega(n/\bar{D})$, that is, $\bar{D} = \Omega(\sqrt{n/B})$, contradicting our assumption that $\bar{D} = o(\sqrt{n/B})$.

Next, consider the problem of finding an approximate count when the diameter is not known in advance. (Our lower bound from Section 4 allows the diameter to be known in advance, but the following requires it to be unknown.) Let N be the best upper bound known in advance on the count. We will show that in order to distinguish a network of size n from a network of size N, nodes a, b must solve a Set Disjointness instance of size $O(n/t_{\mathcal{A}})$, so that again $t_{\mathcal{A}} = \Omega(\sqrt{n/B})$ rounds are required.

Recall that in $G_{s,x,y}$, the distance from any node $(i - 1) \cdot s + 1$ where $i \in X \cap Y$ to nodes a and b is $s > t_{\mathcal{A}}$. Thus, when $X \cap Y \neq \emptyset$, we can choose a node $v := (i-1) \cdot s + 1$ where $i \in X \cap Y$, and "hide" nodes $n+1, \ldots, N$ behind it, adding edges from nodes $n + 1, \ldots, N$ to v and from nodes a, b to nodes $n + 1, \ldots, N$. Let $G'_{s,x,y}$ be the resulting network. Since the distance from node v to nodes a, b exceeds $t_{\mathcal{A}}$, and the new nodes $n + 1, \ldots, N$ are connected only to node v, $t_{\mathcal{A}}$ rounds are insufficient for nodes a, b to distinguish $G_{s,x,y}$ from $G'_{s,x,y}$. Therefore, if $X \cap Y \neq \emptyset$, an algorithm for distinguishing networks of size $n + 2$ from networks of size N cannot terminate by time $t_{\mathcal{A}}$ in $G_{s,x,y}$ (except with small probability). This is sufficient to carry out the reduction from Thm. 5 exactly as before, obtaining an $\Omega(\sqrt{n/B})$ lower bound on any non-trivial approximation of the count.

5.2 A $(D + \tilde{O}(\sqrt{n/B}))$-Round Algorithm for HF$_n$

We now give an algorithm that solves HF$_n$ in nearly-optimal time. If, for example, the minimum input value heard so far is forwarded alongside the messages of our algorithm, this allows nodes to compute the global minimum. The algorithm does not use UIDs, and it only requires an polynomially loose upper bound $N \geq n$ on the count.

High-level overview of the algorithm. Initially, each node computes a sequence of independent Bernoulli variables, and stores the indices of the variables that turned up one. These indices are called *tokens*. The tokens are then forwarded throughout the network by all nodes. If a node does not receive any new tokens for a sufficiently long period of time, it concludes that it has heard from all nodes, and halts. The waiting period is long enough so that if at the end t of the period we do not have $N^t(v) = V$, then during the waiting period the tokens of many new nodes are received by v, and the probability that none of these nodes generated a token that was not previously known is very small.

```
for $k = \lceil \log\log N \rceil, \lceil \log\log N \rceil + 1, \ldots, \lceil \log N \rceil$ do
    Compute independent $X_k^1, \ldots, X_k^{\ell_k} \sim \mathrm{Bernoulli}(2^{-(k+2)})$
    $\mathit{last\_update}_k \leftarrow 0$
$\mathit{Tokens} \leftarrow \{(k,i) \in \mathbb{N}^2 \mid X_k^i = 1\}$, $\mathit{Sent} \leftarrow \emptyset$
for $r = 1, 2, \ldots$ do
    $X \leftarrow$ select the $\beta$ smallest tokens in $\mathit{Tokens} \setminus \mathit{Sent}$
    broadcast $X$ and set $\mathit{Sent} \leftarrow \mathit{Sent} \cup X$
    receive tokens $Y$ from neighbors
    for all $y = (k,i) \in Y \setminus \mathit{Tokens}$ do $\forall k' \geq k : \mathit{last\_update}_{k'} \leftarrow r$
    $\mathit{Tokens} \leftarrow \mathit{Tokens} \cup Y$
    if $\exists k : (|\{(k,i) \in \mathit{Tokens}\}| \leq 2\ell_k/3) \wedge (r - \mathit{last\_update}_k \geq 2\tau_k)$ then halt
```

Algorithm 1. A $(D + \tilde{O}(\sqrt{n/B}))$-round algorithm for HF$_n$

Detailed description. Since nodes do not know the exact size n, we use exponentially-increasing guesses 2^k for $k = \lceil \log\log N \rceil, \ldots, \lceil \log N \rceil$. We refer to each value of k as a *level*. On level k, each node computes ℓ_k independent Bernoulli variables X_k^i with $\Pr[X_i = 1] = 1/2^{k+2}$, where $\ell_k = \tilde{\Theta}(\sqrt{2^k B})$ (the exact value will be fixed later). We denote by $L_k := \sum_{i=1}^{k} \ell_i$ the total number of variables computed on levels $k' \leq k$.

At the beginning of the algorithm, the indices of the variables that turned up one on each level are collected in a set $\mathit{Tokens} = \{(k,i) \mid X_k^i = 1\}$. The tokens are ordered lexicographically — first by level and then by index. Each token can be represented using $\log n + \log\log N$ bits; for simplicity we assume that each message can fit β tokens, that is, $B = \beta(\log n + \log\log N)$ where β is an integer. Pseudocode for the algorithm is given by Algorithm 1. In the sequel, let $\tau_k := \lceil L_k/\beta \rceil$.

After generating an initial set of tokens, the tokens are disseminated in batches of β tokens each, with lower-level tokens taking precedence over higher-level tokens. Each node halts as soon as on some level k, fewer than $2\ell_k/3$ tokens have been received in total, and in the past $2\tau_k = 2\lceil L_k/\beta \rceil$ rounds no new token was received.

The algorithm relies on *pipelining* [18] to quickly disseminate small tokens throughout the network. Because we forward small tokens before large ones, the progress of a token (k,i) can only be impeded by tokens on its own level (k) or lower levels ($k' < k$); there are at most $L_k = \sum_{i=1}^{k} \ell_i$ such tokens, and β of them can be sent per message. Thus the "latency" of token (k,i) is at most $\lceil L_k/\beta \rceil = \tau_k$. More formally, for a set $S \subseteq V$ of nodes, let $A_k(S) := \bigcup_{v \in S} \{(k,i) \mid (k,i) \in \mathit{Tokens}_v(0)\}$ be the level-k tokens generated by the nodes of S. Let $\mathit{Tokens}_v(t)$ stand for the value of the local variable Tokens at node v and time t. The latency of level-k tokens is bounded by the following lemma.

Lemma 4. *For all $v \in V$ and $t \geq \tau_k$, $A_k(N^{t-\tau_k}(v)) \subseteq \mathit{Tokens}_v(t) \subseteq A(N^t(v))$.*

We can now bound the round complexity of the algorithm in terms of the "correct" value of k, which is roughly $\log(n)$.

Lemma 5. *Let $\hat{k} := \min\{\lceil \log\log N \rceil, \lceil \log n \rceil\}$. In graphs of diameter D, the algorithm terminates in $D + 3\lceil L_{\hat{k}}/\beta \rceil$ rounds with probability at least $1 - e^{-\ell_{\hat{k}}/9}$.*

Proof (sketch). It is not difficult to show that the expected number of level-k tokens generated by all the nodes together is at most $\ell_{\hat{k}}/3$. A Chernoff bound shows that w.h.p., the total number of level-k tokens does not exceed $(2/3)\ell_{\hat{k}}$, so the second part of the termination condition is satisfied for $k = \hat{k}$. For the first part of the condition we rely on pipelining: Lemma 4 shows that $A_{\hat{k}}(N^D(v)) \subseteq \mathit{Tokens}_v(D+\tau_k)$ for all nodes v; since $N^D(v) = V$, at time $D + \tau_k$, each node v has already received all tokens generated anywhere in the network. After this time no node can receive any new tokens, so all nodes halt no later than time $D + 3\tau_k$. □

Next we show that w.h.p., nodes do not halt before they have heard from all n nodes.

Lemma 6. *If the level-k termination condition holds at node v at time t, then with probability at least $1 - e^{-\ell_k^2/(3\cdot 2^{k+3}\beta)}$ we have $N^t(v) = V$.*

Proof (sketch). The level-k termination condition asserts that no new level-k tokens are received during the time interval $[t - 2\tau_k, t]$. Assume that $N^t(v) \neq V$, and set $S := N^{t-2\tau_k}(v), S' := N^{t-\tau_k}(v)$. From Lemma 4 we see that

(a) $A_k(S') \subseteq \mathit{Tokens}_v(t)$, that is, all tokens generated by the nodes of S' are known to v at time t; and
(b) $\mathit{Tokens}_v(t-2\tau_k) \subseteq A_k(S)$, i.e., at time $t-2\tau_k$ node v only knows tokens generated by the nodes of S.

Since no new tokens were added to Tokens_v between time $t - 2\tau_k$ and time t, we must have $A_k(S') = A_k(S)$; in other words, the nodes of $S' \setminus S$ did not generate any tokens that were not already generated by the nodes of S. We will show that this is unlikely.

From the level-k termination criterion, at least $\ell_k/3$ tokens were not generated by the nodes of S. Each of these tokens is generated by each node of $S' \setminus S$ with probability $1/2^{k+2}$. Because we assumed that $N^t(v) \neq V$ and the graph is strongly connected, $|S' \setminus S| \geq \tau_k$. Hence, for each token $(k, i) \notin A_k(S)$, we can show that $\Pr[(k,i) \in A_k(S' \setminus S)] \geq \tau_k/2^{k+3}$ independently of the other tokens. It follows that $\Pr[A_k(S') = A_k(S)] \leq \left(1 - \tau_k/2^{k-3}\right)^{\ell_k/3} \leq e^{-\ell_k^2/(3\cdot 2^{k+3}\beta)}$. □

Combining the two lemmas, we see that choosing ℓ_k as $\Theta(\sqrt{2^k \beta \ln N})$ yields a polynomially small probability of any node not halting at time $D + O(L_{\hat{k}}/\beta) = D + \tilde{O}(\sqrt{n/B})$, or halting before it has heard from all n nodes.

Theorem 7. *For any constant c, if $\ell_k \geq \sqrt{3(c+2)\beta \cdot 2^{k+3} \ln N}$, then with probability at least $1 - 1/N^c$ each node v halts at a time $t = D + O(L_k/\beta) = \tilde{O}(D + \sqrt{n/B})$ such that $N^t(v) = V$.*

6 Conclusion

Data aggregation problems are traditionally studied in models that feature symmetric point-to-point communication. However, wireless networks can have *asymmetric* communication topologies, due to the effects of local interference and heterogeneous power assignments. This motivates our interest in directed networks with communication by local broadcast.

Our results show that the traditional strategy of first computing a spanning tree, and then solving various distributed tasks using the tree, is not always optimal for directed networks; for example, while computing a rooted spanning tree can require $\Omega(D + n/B)$ rounds (as we saw in Section 4), certain data aggregates can be computed or approximated in $\tilde{O}(D + \sqrt{n/B})$ rounds. Our lower bounds also imply that it is not possible to quickly compute a small-diameter symmetric spanning subgraph of a directed network with diameter 2. In general it seems that "topology-oblivious" algorithms, such as the algorithm in Section 5.2 and gossip algorithms [6,11], may be better suited for directed networks.

We leave open the question of finding a tight bound on the deterministic time complexity of computing a sensitive function; is there a *deterministic* algorithm that matches the $\Omega(D+\sqrt{n/B})$ lower bound, or can the lower bound be strengthened? For technical reasons, it seems unlikely that a two-party reduction of the style we used in this paper will yield a stronger lower bound, but perhaps multi-party communication complexity lower bounds could be used.

References

1. Awerbuch, B.: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems (detailed summary). In: Proc. 19th ACM Symp. on Theory of Computing (STOC), pp. 230–240 (1987)
2. Chakrabarti, A., Regev, O.: An optimal lower bound on the communication complexity of gap-hamming-distance. In: Proc. 43rd ACM Symp. on Theory of Computing (STOC), pp. 51–60 (2011)
3. Frederickson, G.N., Lynch, N.A.: The impact of synchronous communication on the problem of electing a leader in a ring. In: Proc. 16th ACM Symp. on Theory of Computing (STOC), pp. 493–503 (1984)
4. Indyk, P., Woodruff, D.: Tight lower bounds for the distinct elements problem. In: Proc. 44th IEEE Symp. on Foundations of Computer Science (FOCS), pp. 283–288 (October 2003)
5. Kalyanasundaram, B., Schnitger, G.: The probabilistic communication complexity of set intersection. SIAM J. Discrete Math. 5(4), 545–557 (1992)
6. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 482–491 (2003)
7. Kuhn, F., Locher, T., Schmid, S.: Distributed computation of the mode. In: Proc. 27th ACM Symp. on Principles of Distributed Computing (PODC), pp. 15–24 (2008)
8. Kuhn, F., Locher, T., Wattenhofer, R.: Tight bounds for distributed selection. In: Proc. 19th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), pp. 145–153 (2007)
9. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: Proc. 42nd ACM Symp. on Theory of Computing (STOC), pp. 513–522 (2010)

10. Kushilevitz, E., Nisan, N.: Communication complexity. Cambridge University Press, Cambridge (1997)
11. Mosk-Aoyama, D., Shah, D.: Fast distributed algorithms for computing separable functions. IEEE Transactions on Information Theory 54(7), 2997–3007 (2008)
12. Negro, A., Santoro, N., Urrutia, J.: Efficient distributed selection with bounded messages. IEEE Trans. Parallel and Distributed Systems 8(4), 397–401 (1997)
13. Patt-Shamir, B.: A note on efficient aggregate queries in sensor networks. In: Proc. 23rd ACM Symp. on Principles of Distributed Computing (PODC), pp. 283–289 (2004)
14. Razborov, A.A.: On the distributional complexity of disjointness. Theor. Comput. Sci. 106, 385–390 (1992)
15. Santoro, N., Scheutzow, M., Sidney, J.B.: On the expected complexity of distributed selection. J. Parallel and Distributed Computing 5(2), 194–203 (1988)
16. Santoro, N., Sidney, J.B., Sidney, S.J.: A distributed selection algorithm and its expected communication complexity. Theoretical Computer Science 100(1), 185–204 (1992)
17. Shrira, L., Francez, N., Rodeh, M.: Distributed k-selection: From a sequential to a distributed algorithm. In: Proc. 2nd ACM Symp. on Principles of Distributed Computing (PODC), pp. 143–153 (1983)
18. Topkis, D.M.: Concurrent broadcast for information dissemination. IEEE Trans. Softw. Eng. 11, 1107–1112 (1985)

Black Hole Search with Finite Automata Scattered in a Synchronous Torus

Jérémie Chalopin[1], Shantanu Das[1], Arnaud Labourel[1], and Euripides Markou[2]

[1] LIF, Aix-Marseille University, Marseille, France
{jeremie.chalopin,shantanu.das,arnaud.labourel}@lif.univ-mrs.fr
[2] Department of Computer Science and Biomedical Informatics,
University of Central Greece, Lamia, Greece
emarkou@ucg.gr

Abstract. We consider the problem of locating a black hole in synchronous anonymous networks using finite state agents. A black hole is a harmful node in the network that destroys any agent visiting that node without leaving any trace. The objective is to locate the black hole without destroying too many agents. This is difficult to achieve when the agents are initially scattered in the network and are unaware of the location of each other. In contrast to previous results, we solve the problem using a small team of finite-state agents each carrying a constant number of identical tokens that could be placed on the nodes of the network. Thus, all resources used in our algorithms are independent of the network size.

We restrict our attention to oriented torus networks and first show that no finite team of finite state agents can solve the problem in such networks, when the tokens are not movable, i.e., they cannot be moved by the agents once they have been released on a node. In case the agents are equipped with movable tokens, we determine lower bounds on the number of agents and tokens required for solving the problem in torus networks of arbitrary size. Further, we present a deterministic solution to the black hole search problem for oriented torus networks, using the minimum number of agents and tokens, thus providing matching upper bounds for the problem.

1 Introduction

The exploration of an unknown graph by one or more mobile agents is a classical problem initially formulated in 1951 by Shannon [27] and it has been extensively studied since then (e.g., see [1,8,20]). Recently, the exploration problem has also been studied in unsafe networks which contain malicious hosts of a highly harmful nature, called *black holes*. A black hole is a node which contains a stationary process destroying all mobile agents visiting this node, without leaving any trace. In the *Black Hole Search* problem the goal for the agents is to locate the black hole within finite time. In particular, at least one agent has to survive knowing all edges leading to the black hole. The only way of locating a black hole is to have at least one agent visiting it. However, since any agent visiting a black hole is destroyed without leaving any trace, the location of the black hole must be deduced by some communication mechanism employed by the agents. Four such mechanisms have been proposed in the literature: a) the *whiteboard* model in which there is a whiteboard at each node of the network where the agents can leave messages, b) the *'pure' token* model where the agents carry tokens which they can leave at nodes, c) the *'enhanced' token* model in which the agents can leave tokens at nodes or edges, and d) the time-out mechanism (only for synchronous networks) in which one agent explores a new node while another agent waits for it at a safe node.

The most powerful inter-agent communication mechanism is having whiteboards at all nodes. Since access to a whiteboard is provided in mutual exclusion, this model could also provide the agents a symmetry-breaking mechanism: If the agents start at the same node, they can get distinct identities and then the distinct agents can assign different labels to all nodes. Hence in this model, if the agents are initially co-located, both the agents and the nodes can be assumed to be non-anonymous without any loss of generality.

In asynchronous networks and given that all agents initially start at the same safe node, the Black Hole Search problem has been studied under the whiteboard model (e.g., [9,12,13,14]), the 'enhanced' token model (e.g., [10,15,28]) and the 'pure' token model in [18]. It has been proved that the problem can be solved with a minimal number of agents performing a polynomial number of moves. Notice that in an asynchronous network the number of the nodes of the network must be known to the agents otherwise the problem is unsolvable ([13]). If the graph topology is unknown, at least $\Delta + 1$ agents are needed, where Δ is the maximum node degree in the graph ([12]). Furthermore the network should be 2-connected. It is also not possible to answer the question of *whether* a black hole exists in the network.

In asynchronous networks, with scattered agents (not initially located at the same node), the problem has been investigated for the ring topology ([11,13]) and for arbitrary topologies ([4,19]) in the whiteboard model while in the 'enhanced' token model it has been studied for rings ([16,17]) and for some interconnected networks ([28]).

The consideration of synchronous networks makes a dramatic change to the problem. Now two co-located distinct agents can discover one black hole in any graph by using the time-out mechanism, without the need of whiteboards or tokens. Moreover, it is now possible to answer the question of whether a black hole actually exists or not in the network. No knowledge about the number of nodes is needed. Hence, with co-located distinct agents, the issue is not the feasibility but the time efficiency of black hole search. The issue of efficient black hole search has been studied in synchronous networks without whiteboards or tokens (only using the time-out mechanism) in [5,7,22,23,24] under the condition that all distinct agents start at the same node. However when the agents are scattered in the network, the time-out mechanism is not sufficient anymore.

Indeed the problem seems to be much more difficult in the case of scattered agents and there are very few known results for this scenario. In this paper we study this version of the problem using very simple agents that can be modeled as finite state automata. Our objective is to determine the minimum resources, such as number of agents and tokens, necessary and sufficient to solve the problem in a given class of networks. For the class of ring networks, recent results [3] show that having constant-size memory is not a limitation for the agents when solving this problem. We consider here the more challenging scenario of anonymous torus networks of arbitrary size. We show that even in this case, finite state agents are capable of locating the black hole in all oriented torus networks using only a few tokens. Note that the exploration of anonymous oriented torus networks is a challenging problem in itself, in the presence of multiple identical agents [26]. Since the tokens used by the agents are identical, an agent cannot distinguish its tokens from those of another agent.

While the token model has been mostly used in the exploration of safe networks, the whiteboard model is commonly used in unsafe networks. The 'pure' token model can be implemented with $O(1)$-bit whiteboards, for a constant number of agents and a constant number of tokens, while the 'enhanced' token model can be implemented having a $O(\log d)$-bit whiteboard on a node with degree d. In the whiteboard model, the capacity of each whiteboard is always assumed to be of at least $\Omega(\log n)$ bits, where n is the number of nodes of the network. In all previous papers studying the Black Hole Search problem under a token model apart from [18] and [3], the authors have used the 'enhanced' token model with agents having non-constant memory. The weakest 'pure' token model has been used in [18] for co-located non-constant memory agents equipped with a map in asynchronous networks.

The Black Hole Search problem has also been studied for co-located agents in asynchronous and synchronous directed graphs with whiteboards in [6,24]. In [21] they study the problem in asynchronous networks with whiteboards and co-located agents without the knowledge of incoming link. A different dangerous behavior is studied for co-located agents in [25], where the authors consider a ring and assume black holes with Byzantine behavior, which do not always destroy a visiting agent.

Our Contributions: We consider the problem of locating the black hole in an anonymous but oriented torus containing exactly one black hole, using a team of identical agents that are initially scattered within the torus. Henceforth we will refer to this problem as the BHS problem. We focus our attention on very simple mobile agents. The agents have constant-size memory, they can communicate with other agents only when they meet at the same node and they carry a constant number of identical tokens which can be placed at nodes. The tokens may be movable (i.e. they can be released and picked up later) or unmovable (i.e. they cannot be moved by the agents once they have been released on a node). We prove the following results:

– No finite team of agents can solve the BHS problem in all oriented torus networks using a finite number of *unmovable* tokens.
– For agents carrying any finite number of *movable* tokens, at least three agents are required to solve the problem.

– Any algorithm for solving BHS using 3 agents requires more than one movable token per agent.
– The BHS problem can be solved using three agents and only two movable tokens per agent, thus matching both the lower bounds mentioned above.

In Section 2, we formally describe our model, giving the capabilities of the agents. In Section 3, we prove lower bound on the number of agents and tokens needed to solve the BHS problem in the torus. In Section 4, we present deterministic algorithms for BHS: (i) using $k \geq 3$ agents carrying 3 movable tokens per agents, and (ii) using $k \geq 4$ agents carrying 2 movable tokens per agent. We also present a more involved algorithm that uses exactly 3 agents and 2 tokens per agent thus meeting the lower bounds. All our algorithms are time-optimal and since they do not require any knowledge about the dimensions of the torus, they work in any synchronous oriented torus, using only a finite number of agents having constant-size memory. Due to space limitations, proofs of some of theorems and formal descriptions of the algorithms are omitted and can be found in [2].

2 Our Model

Our model consists of $k \geq 2$ anonymous and identical mobile agents that are initially placed at distinct nodes of an anonymous, synchronous torus network of size $n \times m$, $n \geq 3$, $m \geq 3$. We assume that the torus is oriented, i.e., at each node, the four incident edges are consistently marked as North, East, South and West. Each mobile agent owns a constant number of t identical tokens which can be placed at any node visited by the agent. In all our protocols a node may contain at most three tokens at the same time and an agent carries at most three tokens at any time. A token or an agent at a given node is visible to all agents on the same node, but is not visible to any other agent. The agents follow the same deterministic algorithm and begin execution at the same time and being at the same initial state.

At any single time unit, a mobile agent occupies a node u of the network and may 1) detect the presence of one or more tokens and/or agents at node u, 2) release/take one or more tokens to/from the node u, and 3) decide to stay at the same node or move to an adjacent node. We call a token *movable* if it can be moved by any mobile agent to any node of the network, otherwise we call the token *unmovable* in the sense that, once released, it can occupy only the node in which it has been released.

Formally we consider a mobile agent as a finite Moore automaton $\mathcal{A} = (\mathcal{S}, S_0, \Sigma, \Lambda, \delta, \phi)$, where $\mathcal{S}$ is a set of $\sigma \geq 2$ states; S_0 is the *initial* state; Σ is the set of possible configurations an agent can see when it enters a node; $\delta : \mathcal{S} \times \Sigma \rightarrow \mathcal{S}$ is the transition function; and $\phi : \mathcal{S} \rightarrow \Lambda$ is the output function. Elements of Σ are quadruplets (D, x, y, b) where $D \in \{\texttt{North}, \texttt{South}, \texttt{East}, \texttt{West}, \texttt{none}\}$ is the direction through which the agent has arrived at the node, x is the number of tokens (at most 3) at that node, y is number of tokens (at most 3) carried by the agent and $b \in \{true, false\}$ indicates whether there is at least another agent at the node or not. Elements of Λ are quadruplets

(P, s, X, M) where $P \in \{\texttt{put}, \texttt{pick}\}$ is the action performed by the agent on the tokens, $s \in \{0, 1, 2, 3\}$ is the number of tokens concerned by the action A, $X \in \{\texttt{North}, \texttt{South}, \texttt{East}, \texttt{West}, \texttt{none}\}$ is the edge marked as dangerous by the agent, and $M \in \{\texttt{North}, \texttt{South}, \texttt{East}, \texttt{West}, \texttt{none}\}$ is the move performed by the agent. Note that the agent always performs the action before the marking and the marking before the move.

Note that all computations by the agents are independent of the size $n \times m$ of the network since the agents have no knowledge of n or m. There is exactly one black hole in the network. An agent can start from any node other than the black hole and no two agents are initially co-located. Once an agent detects a link to the black hole, it marks the link permanently as dangerous (i.e., disables this link). Since the agents do not have enough memory to remember the location of the black hole, we require that at the end of a black hole search scheme, all links incident to the black hole (and only those links) are marked dangerous and that there is at least one surviving agent. Thus, our definition of a successful BHS scheme is slightly different from the original definition. The time complexity of a BHS scheme is the number of time units needed for completion of the scheme, assuming the worst-case location of the black hole and the worst-case initial placement of the scattered agents.

3 Impossibility Results

In this section we give lower bounds on the number of agents and the number and type of tokens needed for solving the BHS problem in any anonymous, synchronous and oriented torus.

3.1 Agents with Unmovable Tokens

We will prove that any constant number of agents carrying a constant number of unmovable tokens each, can not solve the BHS problem in an oriented torus. The idea of the proof is the following: We show that an adversary (by looking at the transition function of an agent) can always select a big enough torus and initially place the agents so that no agent visits nodes which contain tokens left by another agent, or meets with another agent. Moreover there are nodes on the torus never visited by any agent. Hence the adversary may place the black hole at a node not visited by any of the agents to make the algorithm fail. This result is based on ideas presented earlier in [26].

Theorem 1. *For any constant numbers k, t, there exists no algorithm that solves BHS in all oriented tori containing one black hole and k scattered agents, where each agent has a constant memory and t unmovable tokens.*

3.2 Agents with Movable Tokens

We show the following impossibility results.

Lemma 1. *Two agents carrying any number of movable tokens cannot solve the BHS problem in an oriented torus even if the agents have unlimited memory.*

Proof. Assume w.l.o.g. that the first move of the agents is going East. Suppose that the black hole has been placed by an adversary at the East neighbor of an agent. This agent vanishes into the black hole after its first move. The adversary places the second agent such that it vanishes into the black hole after its first vertical move, or it places the agent in a horizontal ring not containing the black hole if the agent never performs vertical moves. Observe that the trajectories of the two agents intersect only at the black hole and neither can see any token left by the other agent. Neither of the agents will ever visit the East neighbor of the black hole and thus, they will not be able to correctly mark all links incident to the black hole.

Thus, at least three agents are needed to solve the problem. We now determine a lower bound on the number of tokens needed by three scattered agents to solve BHS.

Lemma 2. *There exists no algorithm that could solve the BHS problem in all oriented tori using three agents with constant memory and one movable token each.*

Proof. (Sketch) Clearly, in view of Theorem 1, an algorithm which does not instruct an agent to leave its token at a node, cannot solve the BHS problem. Hence any potentially correct algorithm should instruct an agent to leave its token down. Moreover this decision has to be taken after a finite number of steps (due to agents' constant memory). After that the agents visit new nodes until they see a token. We can show that if the agents visit only a constant number of nodes before returning to meet their tokens they cannot visit all nodes of the torus. If they move their tokens each time they see them and repeat the previous procedure (i.e., visit a constant number of nodes and return to meet their tokens), we can show that they will find themselves back at their initial locations and initial states without having met with other agents and leaving some nodes unvisited. An adversary may place the black hole at an unvisited node to make the algorithm fail. Now consider the case that at some point an algorithm instructs the agents to visit a non-constant number of nodes until they see a token (e.g., leave your token down and go east until you see a token). Again we can show that an adversary may initially place the agents and the black hole, and select the size of the torus so that two of the agents enter the black hole without leaving their tokens close to it: The agent (say A) that enters first into the black hole has been initially placed by an adversary so that it left its token more than a constant number of nodes away from the black hole. The adversary initially places another agent B so that it enters the black hole before it meets A's token. Furthermore B leaves its token more than a constant number of nodes away from the black hole. Hence the third agent, even if it meets the tokens left by A or B, it could not decide the exact location of the black hole.

Theorem 2. *At least three agents are necessary to solve the BHS problem in an oriented torus of arbitrary size. Any algorithm solving this problem using three agents requires at least two movable tokens per agent.*

4 Algorithms for BHS in a Torus using Movable Tokens

Due to the impossibility result from the previous section, we know that unmovable tokens are not sufficient to solve BHS in a torus. In the following, we will use only movable tokens. To explore the torus an agent uses the Cautious-Walk technique [13] using movable tokens. In our algorithms, a *Cautious-Walk in direction D with x tokens* means the following (see Procedure 1): (i) the agent releases a sufficient number of tokens such that there are exactly x tokens at the current node, (ii) the agent moves one step in direction D and if it survives, the agent immediately returns to the previous node, (iii) the agent picks up the tokens released in step (i) and again goes one step in direction D. If an agent vanishes during step (ii), any other agent arriving at the same location sees x tokens and realizes a potential danger in direction D. Depending on the algorithm an agent may use 1, 2, or 3 tokens to implement the Cautious-Walk.

Procedure `Cautious-walk(`*Direction D, integer x*`)`

```
  /* Procedure used by the agent to explore the nodes        */
1 Put tokens until there are $x$ tokens;
2 **Go** one step along $D$ and then go back;
  /* test if the node in direction $D$ is the black hole     */
3 Pick up the tokens released in the first step;
4 **Go** one step along $D$;
```

4.1 Solving BHS Using $k \geq 3$ Agents and 3 Tokens

We show that three agents suffice to locate the black hole if the agents are provided with three tokens using the algorithm presented below.

Algorithm BHS-torus-33
An agent explores one horizontal ring at a time and then moves one step South to the next horizontal ring and so on. When exploring a horizontal ring, the agent leaves one token on the starting node. This node is called the *homebase* of the agent and the token left (called homebase token) is used by the agent to decide when to proceed to the next ring. The agent then uses the two remaining tokens to repeat Cautious-Walk in the East direction until it has seen twice a node containing one token. Any node containing one token is a homebase either of this agent or of another agent. The agent moves to the next horizontal ring below after encountering two homebases. However before moving to the next ring, it does a cautious walk in the South direction with three tokens (the two tokens it carries plus the homebase token). If the agent survives and the node reached by the agent has one token, the agent repeats a cautious walk in the East

direction (with two tokens) until it reaches an empty node. The agent can now use this empty node as its new homebase. It then repeats the same exploration process for this new ring leaving one token at its new homebase.

Whenever the agent sees two or three tokens at the end of a cautious-walk, the agent has detected the location of the black hole: If there are two (resp. three) tokens at the current node, the black hole is the neighboring node w to the East (resp. South). In this case, the agent stops its normal execution and then traverses a cycle around node w, visiting all neighbors of w and marking all the links leading to w as dangerous.

Theorem 3. *Algorithm BHS-torus-33 correctly solves the BHS problem with* 3 *or more agents.*

Proof. An agent may visit an unexplored node only while going East or South. If one agent enters the black hole going East (resp. South), there will be two (resp. three) tokens on the previous node and thus, no other agent would enter the black hole through the same link. This implies that at least one agent always survives. Whenever an agent encounters two or three tokens at the end of a Cautious-Walk, the agent is certain of the location of the black hole since any alive agent would have picked up its Cautious-Walk tokens in the second step of the cautious walk (The agents move synchronously always using cautious walk and taking three steps for each move).

4.2 BHS Using $k \geq 4$ Agents and 2 Tokens Each

We now present an algorithm that uses only two tokens per agent, but requires at least 4 agents to solve BHS.

During the algorithm, the agents put two tokens on a node u to signal that either the black hole is on the South or the East of node u. Eventually, both the North neighbor and the West neighbor of the black hole are marked with two tokens. Whenever there is a node v such that there are exactly two tokens at both the West neighbor of v and the North neighbor of v, then we say that there exists a *Black-Hole-Configuration* (BHC) at v.

Algorithm BHS-torus-42

The agent puts two tokens on its starting node (homebase). It then performs a Cautious-Walk in the East direction. If the agent survives, it returns, picks up one token and repeats the Cautious-Walk with one token in the East direction (leaving the other token on the homebase) until it reaches a node containing one or two tokens.

- If the agent reaches a node u containing two tokens, then the black hole is the next node on the East or on the South (See Property C of Proposition 1). The agent stops its exploration and checks whether the black hole is the East neighbor or the South neighbor.
- Whenever an agent reaches a node containing one token, it performs a Cautious-Walk in East direction with two tokens and then continues the

Cautious-Walk in the same direction with one token. If the agent encounters three times[1] a node containing one token, it moves to the next horizontal ring below. To do that it first performs a Cautious-Walk with two tokens in the South direction. If the agent survives and reaches the ring below, it can start exploring this horizontal ring. If the current node is not empty, the agent repeats a cautious walk with two token in the East direction until it reaches an empty node. Now the agent repeats the same exploration process using the empty node as its new homebase. Whenever the agent encounter a node with two tokens, it stops its exploration and checks whether the black hole is the East or South neighbor.

In order to check if the black hole is the East neighbor v or the South neighbor w of the current node u (containing two tokens), the agent performs the following actions: The agent reaches the West neighbor x of w in exactly three time units by going West and South (and waiting one step in between). If there are two tokens on this node x then w is the black hole. Otherwise, the agent performs a cautious walk in the East direction with one token (or with two tokens if there is already one token on node x). If it safely reaches node w, then the black hole is the other node v. Otherwise the agent would have fallen into the black hole leaving a BHC at node w. An agent that discovers the black hole, traverses a cycle around the black hole visiting all its neighbors and marking all the links leading to it as dangerous.

Proposition 1. *During an execution of BHS-torus-42 with $k \geq 4$ agents, the following properties hold:*

- *A When an agent checks the number of tokens at a node, all surviving agents have picked up their cautious-walk token.*
- *B At most three agents can enter the black hole:*
  - *(a) at most one agent going South leaving two tokens at the previous node.*
  - *(b) at most two agents going East, each leaving one of its tokens at the previous node.*
- *C When an agent checks the number of tokens at a node, if there are two tokens then the black hole is either the East or the South neighbor of the current node.*
- *D After an agent starts exploring a horizontal ring, one of the following eventually occurs:*
  - *(a) If this ring is safe, the agent eventually moves to the next horizontal ring below.*
  - *(b) Otherwise, either all agents on this ring fall into the black hole or one of these agents marks all links to the black hole.*

Theorem 4. *Algorithm BHS-torus-42 correctly solves the black hole search problem with $k \geq 4$ agents, each having two tokens.*

[1] The agent may encounter homebases of two agents which have both fallen into the black hole. (In this case it must continue in the same direction until it locates the black hole).

Proof. Property B of Proposition 1 guarantees that at least one agent never enters the black hole. Property D ensures that one of the surviving agents will identify the black hole. Property C shows that if the links incident to a node w are marked as dangerous by the algorithm, then node w is the black hole.

4.3 BHS with 3 Agents and 2 Movable Tokens

Using the techniques presented so far, we now present the algorithm that uses exactly 3 agents and two tokens per agent. The algorithm is quite involved and we present here only the main ideas of the algorithm. The complete algorithm along with a proof of correctness can be found in [2]. Notice first that we can not prevent two of the three agents from falling into the black hole (see proof of Lemma 1). To ensure that no more than two agents enter the black hole, the algorithm should allow the agent to move only in two of the possible four directions (when visiting unexplored nodes). When exploring the first horizontal ring, an agent always moves in the East direction, using a Cautious-Walk as before and keeping one token on the starting node (homebase). This is called procedure *First-Ring*. Once an agent has completely explored one horizontal ring, it explores the ring below, using procedure *Next-Ring*. During procedure *Next-Ring*, an agent traverses the already explored ring and at each node u of this ring, the agent puts one token, traverses one edge South (to check the node just below node u), and then immediately returns to node u and picks up the token. Note that an agent may fall into the black hole only when going South during procedure *Next-Ring* or when going East during procedure *First-Ring*. We ensure that at most one agent falls into the black hole from each of these two directions. The surviving agent must then identify the black hole from the tokens left by the other agents.

Algorithm 2. BHS-Torus-32

```
  /* Algorithm for BHS in Oriented Torus (3 agents, 2 tokens)      */
1 First-Ring;
2 repeat
3    Init-Next-Ring;
4    Next-Ring;
5 until until black hole is found ;
```

For this algorithm, we redefine the *Black-Hole-Configuration* (BHC) as follows: If there is a node v such that there is one or two tokens at both the West neighbor of v and the North neighbor of v, then we say that a BHC exists at v. The algorithm should avoid forming a black hole configuration at any other node except the black hole. In particular, when the agents put tokens on their homebase, these tokens should not form a BHC. This requires coordination between any two agents that are operating close to each other (e.g. in adjacent rings of the torus) and it is not always possible to ensure that a BHC is never formed at a safe node.

The main idea of the algorithm is to make two agents meet whenever they are close to each other (this requires a complicated synchronization and checking procedure). If any two agents manage to gather at a node, we can easily solve BHS using the standard procedure for colocated agents[2] with the time-out mechanism (see [3,5]) . On the other hand, if the agents are always far away from each other (i.e. more than a constant distance) then they do not interfere with the operations of each other until one of them falls into the black hole. The agents explore each ring, other than the first ring, using procedure *Next-Ring*. We have another procedure called *Init-Next-Ring* that is always executed at the beginning of *Next-Ring*, where the agents check for certain special configurations and take appropriate action. If the tokens on the potential homebases of two agents would form a BHC on a safe node, then we ensure two agents meet.

Synchronization

During the algorithm, we ensure that two agents meet if they start the procedure *Init-Next-Ring* from nearby locations. We achieve this by keeping the agents synchronized to each other, in the sense that they start executing the procedure at the same time, in each iteration. More precisely, we ensure the following property:

Property 1. When one agent starts procedure *Init-Next-Ring*, any other surviving agent either (i) starts procedure *Init-Next-Ring* at exactly the same time, or (ii) waits in its current homebase along with both its tokens during the time the other agent is executing the procedure or, (iii) has not placed any tokens at its homebase.

Notice that if there are more than one agent initially located at distinct nodes within the same horizontal ring, an agent cannot distinguish its homebase from the homebase of another agent, and thus an agent would not know when to stop traversing this ring and go down to the next one. We get around this problem by making each agent traverse the ring a sufficient number of times to ensure that every node in this ring is explored at least once by this agent. To be more precise, each agent will traverse the ring until it has encountered a homebase node six times (recall that there can be either one, two or three agents on the same ring). This implies that in reality the agent may traverse the same ring either twice, or thrice, or six times. If either all agents start in distinct rings or if all start in the same ring then, the agents would always be synchronized with each other (i.e. each agent would start the next iteration of *Next-Ring* at the same time). The only problem occurs when two agents start on the same ring and the third agent is on a different ring. In this case, the first two agents will be twice as fast as the third agent. We introduce waiting periods appropriately to ensure that Property 1 holds.

For both the procedures *First-Ring* and *Next-Ring*, we define one *big-step* to be the period between when an agent arrives at a node v from the West with

[2] Note that the agents meeting at a node can be assigned distinct identifiers since they would arrive from different directions.

its token(s) and when it has left node v to the East with its token(s). During a big-step, the agent would move to an unsafe node (East or South), come back to pick its token, wait for some time at v before moving to the next node with its token. The waiting period is adjusted so that an agent can execute the whole procedure *Init-Next-Ring* during this time. Thus, the actual number of time units for each big-step is a fixed constant D.

Algorithm *BHS-Torus-32*

Procedure *First-Ring*
During this procedure the agent explores the horizontal ring that contains its starting location. The agent puts one token on its homebase and uses the other token to perform cautious-walk in the direction East, until it enters a node with some tokens. If it finds a node with two tokens then the next node is the black hole. Thus, the agent has solved BHS. Otherwise, if the agent finds a node with a single token this is the homebase of some agent (maybe itself). The agent puts the second token on this node and continues the walk without any token (i.e. it imitates the cautious-walk). If it again encounters a node with a single token, then the next node is the black hole and the algorithm terminates. Otherwise, the agent keeps a counter initialized to one and increments the counter whenever it encounters a node containing two tokens. When the counter reaches a value of six, the procedure terminates. At this point the agent is on a node with two tokens (which it can use for the next procedure).

Unless an agent enters or locates the black hole, the procedure *First-Ring* requires exactly $6nD$ time units for an agent that is alone in the ring, $3nD$ for two agents that start on the same ring, and $2nD$ if all the three agents start on the same ring.

Procedure *Init-Next-Ring*
An agent executes this procedure at the start of procedure *Next-Ring* in order to choose its new homebase for exploring the next ring. The general idea is that the agent checks the node u on the South of its current location, move its two tokens to the East, and then goes back to u. If there is another agent that has started *Next-Ring* on the West of u (i.e., without this Procedure, the homebases of the two agents would have formed a BHC), the agents can detect it, and *Init-Next-Ring* is designed in such a way that the two agents meet. More precisely, when an agent executes *Init-Next-Ring* on horizontal ring i without falling into the black hole, we ensure that either (i) it meets another agent, or (ii) it locates the black hole, or (iii) it detects that the black hole is on ring $i + 2$, or (iv) the token it leaves on its homebase does not form a BHC with a token on ring $i + 1$. In case (iii), the agent executes *Black-Hole-in-Next-Ring*; in case (iv), it continues the execution of *Next-Ring*.

Procedure *Next-Ring*
The agent keeps one token on the homebase and with the other token performs a special cautious-walk during which it traverses the safe ring and at each node

it puts a token, goes South to check the node below, returns back and moves the token to the East. The agent keeps a counter initialized to zero, which it increments whenever it sees a node with a token on the safe ring. When the agent sees a token on the safe ring, it does not go South, since this may be dangerous. Instead, the agent goes West and South, and if it does not see any token there, it puts a token and goes East. If the agent enters the black hole, it has left a BHC. When the counter reaches a value of six, the procedure terminates.

During the procedure, an agent keeps track of how many (1 or 2) tokens it sees in the safe ring and the ring below. This information is stored as a sequence (of length at most 24). At the end of the procedure, using this sequence, an agent in the horizontal ring i can detect whether (i) the Black hole lies in the horizontal ring $i+1$ or $i+2$, or, (ii) there are two other agents in the ring i and ring $i+1$ respectively, or, (iii) the ring $i+1$ does not contain the black hole. In scenario (i), the agent executes procedure *Black-Hole-in-Next-Ring*; in scenario (ii), the agent meets with the other agent in the same ring; in scenario (iii), the agent moves to the next horizontal ring (i.e. ring $i+1$) to start procedure *Init-Next-Ring* again.

Procedure *Black-Hole-in-Next-Ring*
The agent executes this procedure only when it is certain that the black hole lies in the ring below its current position. The procedure is similar to *Next-Ring*; the main difference being that the agent does not leave a homebase token. During the procedure, either (i) the agent detects the location of the black hole and marks all links to the black hole or (ii) the agent falls into the black hole, forming a BHC at the black hole.

Theorem 5. *Algorithm* BHS-Torus-32 *correctly solves the BHS problem in any oriented torus with exactly three agents carrying two tokens each.*

5 Conclusions

We showed that at least three agents are needed to solve BHS in oriented torus networks and these three agents must carry at least two movable tokens each for marking the nodes. The algorithm BHS-Torus-32 uses the smallest possible team of agents (i.e., 3) carrying the minimum number of tokens (i.e., 2) and thus, it is optimal in terms of resource requirements. However, on the downside this algorithm works only for $k = 3$ agents. In combination with algorithm BHS-Torus-42 (which solves the problem for any $k \geq 4$ agents carrying 2 tokens each), these algorithms can solve the black hole search problem for any $k \geq 3$, if the value of k is known. Unfortunately, algorithms BHS-Torus-32 and BHS-Torus-42 cannot be combined to give an algorithm for solving the BHS problem for any $k \geq 3$ agents without the knowledge of k: Algorithm BHS-Torus-32 for 3 agents will not correctly locate the black hole if the agents are more than 3, while in the algorithm BHS-Torus-42, 3 of the agents may fall into the black hole. Hence, whether the problem can be solved for $k \geq 3$ agents equipped with 2 tokens, without any knowledge of k, remains an interesting (and we believe challenging) open question. Another interesting open problem is to determine the minimum size of a team of agents carrying one token each, that can solve the BHS problem. Note that the impossibility result for three agents carrying one token each, does not immediately generalize to the case of 4 or more agents, as in those cases, we cannot exclude the possibility that two surviving agents manage to meet.

It is interesting to compare our results with the situation in a synchronous, oriented, anonymous ring, which can be seen as a one dimensional torus ([3]): The minimum trade-offs between the number of agents and the number of tokens, in this case, are 4 agents with 2 unmovable tokens or 3 agents with 1 movable token each. Additionally, in an *unoriented* ring the minimum trade-offs are 5 agents with 2 unmovable tokens or 3 agents with 1 movable token each whereas the situation in an unoriented torus has not been studied. Hence another open problem is solving the BHS problem in a d-dimensional torus, $d > 3$, as well as in other network topologies.

References

1. Bender, M.A., Slonim, D.: The power of team exploration: Two robots can learn unlabeled directed graphs. In: Proceedings of 35th Annual Symposium on Foundations of Computer Science, pp. 75–85 (1994)
2. Chalopin, J., Das, S., Labourel, A., Markou, E.: Black hole search with finite automata scattered in a synchronous torus. arxiv:1106.6037 (2011)
3. Chalopin, J., Das, S., Labourel, A., Markou, E.: Tight bounds for scattered black hole search in a ring. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 186–197. Springer, Heidelberg (2011)
4. Chalopin, J., Das, S., Santoro, N.: Rendezvous of mobile agents in unknown graphs with faulty links. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 108–122. Springer, Heidelberg (2007)
5. Cooper, C., Klasing, R., Radzik, T.: Searching for black-hole faults in a network using multiple agents. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 320–332. Springer, Heidelberg (2006)
6. Czyzowicz, J., Dobrev, S., Královič, 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)
7. Czyzowicz, J., Kowalski, D., Markou, E., Pelc, A.: Searching for a black hole in synchronous tree networks. Combinatorics, Probability & Computing 16(4), 595–619 (2007)
8. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. Journal of Graph Theory 32(3), 265–297 (1999)
9. Dobrev, S., Flocchini, P., Kralovic, R., Prencipe, G., Ruzicka, P., Santoro, N.: Optimal search for a black hole in common interconnection networks. Networks 47(2), 61–71 (2006)
10. Dobrev, S., Flocchini, P., Kralovic, R., Santoro, N.: Exploring a dangerous unknown graph using tokens. In: Proceedings of 5th IFIP International Conference on Theoretical Computer Science, pp. 131–150 (2006)
11. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Multiple agents rendezvous in a ring in spite of a black hole. In: Proceedings of 6th International Conference on Principles of Distributed Systems, pp. 34–46 (2003)

12. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Searching for a black hole in arbitrary networks: Optimal mobile agents protocols. Distributed Computing 19(1), 1–19 (2006)
13. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Mobile search for a black hole in an anonymous ring. Algorithmica 48, 67–90 (2007)
14. Dobrev, S., Flocchini, P., Santoro, N.: Improved bounds for optimal black hole search in a network with a map. In: Proceedings of 10th International Colloquium on Structural Information and Communication Complexity, pp. 111–122 (2004)
15. Dobrev, S., Královič, R., Santoro, N., Shi, W.: Black hole search in asynchronous rings using tokens. In: Calamoneri, T., Finocchi, I., Italiano, G.F. (eds.) CIAC 2006. LNCS, vol. 3998, pp. 139–150. Springer, Heidelberg (2006)
16. Dobrev, S., Santoro, N., Shi, W.: Scattered black hole search in an oriented ring using tokens. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium, pp. 1–8 (2007)
17. Dobrev, S., Santoro, N., Shi, W.: Using scattered mobile agents to locate a black hole in an un-oriented ring with tokens. International Journal of Foundations of Computer Science 19(6), 1355–1372 (2008)
18. Flocchini, P., Ilcinkas, D., Santoro, N.: Ping pong in dangerous graphs: Optimal black hole search with pure tokens. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 227–241. Springer, Heidelberg (2008)
19. Flocchini, P., Kellett, M., Mason, P., Santoro, N.: Map construction and exploration by mobile agents scattered in a dangerous network. In: Proceedings of IEEE International Symposium on Parallel & Distributed Processing, pp. 1–10 (2009)
20. Fraigniaud, P., Gasieniec, L., Kowalski, D., Pelc, A.: Collective tree exploration. Networks 48, 166–177 (2006)
21. Glaus, P.: Locating a black hole without the knowledge of incoming link. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 128–138. Springer, Heidelberg (2009)
22. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Hardness and approximation results for black hole search in arbitrary graphs. Theoretical Computer Science 384(2-3), 201–221 (2007)
23. Klasing, R., Markou, E., Radzik, T., Sarracco, F.: Approximation bounds for black hole search problems. Networks 52(4), 216–226 (2008)
24. Kosowski, A., Navarra, A., Pinotti, C.: Synchronization helps robots to detect black holes in directed graphs. In: Proceedings of 13th International Conference on Principles of Distributed Systems, pp. 86–98 (2009)
25. Kràlovič, R., Miklík, S.: Periodic data retrieval problem in rings containing a malicious host. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 157–167. Springer, Heidelberg (2010)
26. Kranakis, E., Krizanc, D., Markou, E.: Deterministic symmetric rendezvous with tokens in a synchronous torus. Discrete Applied Mathematics 159(9), 896–923 (2011)
27. Shannon, C.E.: Presentation of a maze-solving machine. In: Proceedings of 8th Conference of the Josiah Macy Jr. Found (Cybernetics), pp. 173–180 (1951)
28. Shi, W.: Black hole search with tokens in interconnected networks. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 670–682. Springer, Heidelberg (2009)

Synchronous Rendezvous for Location-Aware Agents

Andrew Collins[1], Jurek Czyzowicz[2,⋆], Leszek Gąsieniec[1,⋆⋆], Adrian Kosowski[3], and Russell Martin[1]

[1] University of Liverpool, Liverpool L69 3BX, UK
{A.P.Collins,L.A.Gasieniec,Russell.Martin}@liverpool.ac.uk
[2] Département d'informatique, Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada
jurek@uqo.ca
[3] INRIA Bordeaux Sud-Ouest, LaBRI, 33405 Talence, France
kosowski@labri.fr

Abstract. We study rendezvous of two anonymous agents, where each agent knows its own initial position in the environment. Their task is to meet each other as quickly as possible. The time of the rendezvous is measured by the number of synchronous rounds that agents need to use in the worst case in order to meet. In each round, an agent may make a simple move or it may stay motionless. We consider two types of environments, finite or infinite graphs and Euclidean spaces. A simple move traverses a single edge (in a graph) or at most a unit distance (in Euclidean space). The rendezvous consists in visiting by both agents the same point of the environment simultaneously (in the same round).

In this paper, we propose several asymptotically optimal rendezvous algorithms. In particular, we show that in the line and trees as well as in multi-dimensional Euclidean spaces and grids the agents can rendezvous in time $\mathcal{O}(d)$, where d is the distance between the initial positions of the agents.

The problem of location-aware rendezvous was studied before in the asynchronous model for Euclidean spaces and multi-dimensional grids, where the emphasis was on the length of the adopted rendezvous trajectory. We point out that, contrary to the asynchronous case, where the cost of rendezvous is dominated by the size of potentially large neighborhoods, the agents are able to meet in all graphs of at most n nodes in time almost linear in d, namely, $\mathcal{O}(d \log^2 n)$. We also determine an infinite family of graphs in which synchronized rendezvous takes time $\Omega(d)$.

1 Introduction

In the *rendezvous problem* two mobile agents have to meet, starting at different locations of the environment. Each agent is unaware of the location of the other, hence it must use some procedure to explore (a part of) the environment in order to find the other agent. We consider rendezvous setting in which the main goal is to minimize the time required by the agents to meet as a function of their initial distance. The rendezvous problem has been extensively studied in the literature. For example, two comprehensive

⋆ Supported in part by NSERC discovery grant.
⋆⋆ Supported in part by the Royal Society IJP grant.

surveys by Alpern [1,3] and the book by Alpern and Gal [4] present numerous efficient randomized rendezvous strategies. The rendezvous process can be presented as a search game, when two agents collaborate against an adversary, which tries to prevent or delay their meeting. The rendezvous problem can be also seen a version of the consensus problem in which agents have to agree on the meeting point and time [27].

In general, the results in the domain can be classified as those adopting some *geometric setting*, e.g., rendezvous in the line [9,10,22], or in the plane [5,6,29,28]), and those considering rendezvous in graphs, e.g., [1,2]. In the case of unlabeled graphs, a deterministic solution to the rendezvous problem requires breaking symmetry, which may be done by marking the nodes of the graph using pebbles or whiteboards (e.g., see [10,25,26]), using labeled agents (e.g. [17,23]) or exploring asymmetric positions of the agents in the graph (e.g., [14]).

In this paper we use yet another method of breaking symmetry by providing to each agent information about its own position in the environment (this implies that the environment may not be unknown to the agent). The assumption that the distributed agents know their initial location in the geometric environment was considered in the past in the context of geometric routing, e.g., [24], where it was assumed that the agent knows its own position as well as the position of the destination, or broadcasting, (cf. [19,20]), where the position awareness of the broadcasting node only was 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.

In [13,8] the authors study the rendezvous problem of location-aware agents in the asynchronous case. Previously, the asynchronous rendezvous was studied in [16] for lines and rings, while for arbitrary graphs [16] gave the exponential-time rendezvous procedure, under the condition that the bound on the size of the graph is known to the agent. This condition was suppressed in [15], where feasibility of asynchronous rendezvous was settled for arbitrary (even infinite) graphs and geometric environments. Both approaches in [16] and [15]) lead to very inefficient, exponential-time rendezvous algorithms for labeled agents. However, the authors of [13] introduced the concept of covering sequences that permitted location aware agents to meet along the route of polynomial length in d in multi-dimensional grids. Their result was further advanced in [8], where the proposed algorithm provides a route, leading to rendezvous, of length being only a polylogarithmic factor away from the optimal rendezvous trajectory. The inherent bottleneck, however, in asynchronous location-aware rendezvous is in potentially large volume of local neighborhoods. In the worst case, every agent must search through its entire neighborhood of radius d when, e.g., the other agent is immobilized by the adversary that controls the actions of both agents.

The main emphasis in this paper is on *local rendezvous*, i.e., the agents are expected to meet without visiting remote parts of the network. In this setting the rendezvous cost tends to be proportional to d, where d is the distance between the initial positions of the two agents or, in the case of gathering, d refers to the diameter of the ball that covers all agents prompted to meet. Some *local rendezvous* strategies were studied in the geometric setting where the agents have either a complete or limited visibility [28]. E.g., in [7] Ando *et al.* studied convergence stability of multiple agents represented as points on the plane. In a similar geometric setting Cohen and Peleg considered convergence properties of gravitational algorithms including scenarios populated by crash faults [12].

The synchronous deterministic rendezvous for labeled agents in graphs was first studied in [17], where the main result was a polynomial-time rendezvous algorithm. The authors of [23] and [30] independently extended the approach of [17] to the case of agents starting their movement at arbitrary delay. However the algorithms from [17,23,30] are highly polynomial in the size of the network. Thanks to the location-awareness assumption, the approach used in our paper results in very efficient algorithms, linear or slightly super-linear in the initial distance d between the agents, working, in some cases, also for infinite graphs and multi-dimensional grids and spaces.

The Model. In this paper we assume that the network environment is represented by a graph (finite or infinite) in which mobile agents visit nodes by traversing edges in both directions. The approach is also extended to the continuous, geometric setting. It is assumed that all agents move with the uniform speed. More precisely, a mobile agent requires a unit of time to traverse an edge of the graph, or a segment of length 1 in the geometric setting. Each agent can access its clock and agents' clocks are synchronized to tick at the same time moments. An agent can count the number of rounds (clock ticks) and use this information to plan its actions. We assume that both agents start their actions simultaneously, i.e., both clocks indicate the same time. Agents are anonymous. However, each agent is aware of its initial location that can be adopted as its unique label. We also assume that the agents are fully aware of the network topology. The agents can meet each other on vertices of the graph, and they can pass through each other without meeting, while traversing an edge in opposite directions. We focus on deterministic rendezvous procedures (i.e., no randomization is allowed). The movement of an agent is determined by its current location and the time on its clock.

Please note that due to space restrictions several proofs are omitted in this version of the paper.

2 Linear Time Rendezvous on the Infinite Line

The first case we consider here refers to the infinite line $L = (-\infty, +\infty)$. Recall that the mobile agents are aware of their initial location on the line and a sense of (positive or negative) direction. Note that in this setting it is easy to meet agents at, say, the origin of the line. This, however, will not guarantee local rendezvous. I.e., the agents may have to traverse a very long distance, much longer than the distance d that separates the initial positions of the agents. Instead, the agents travel in a *zig-zag* fashion at increasing distances. For agents starting at distance d, this guarantees rendezvous in time $\mathcal{O}(d)$.

We first assume that all agents are initially situated at integer points on the line, and all begin the rendezvous procedure at time $t = 0$. Later, we also discuss the case of non-integer starting points.

The rendezvous proceeds in a series of *iterations*. Each iteration with index $i \geq 1$ consists of two stages: stage 1 and stage 2. We set $\ell_i = 2^{i-2}$, for $i \geq 2$. Informally speaking, during iteration i, the agents are initially divided into *odd* and *even* groups located at a distance of $2\ell_i$ from each other. The *odd* agents move to the right a distance of ℓ_i (stage 1), then to the left a distance of $2\ell_i$ (stage 2). The *even* agents move left a distance of ℓ_i (stage 1), then right a distance of $2\ell_i$ (stage 2). Thus, groups of agents meet both of their neighboring groups, and then some of the groups will merge at the end of the iteration.

More formally, let us define $F_1^c(k) = \{k\}$ for each integer k. We demonstrate the following invariant. During each iteration $i = 1, 2, 3, \ldots$, the agents are partitioned according to their initial positions on the line into the following groups at the end of stages 1 and 2:

1. $F_i^h(k) = \{k \cdot 2^i - 2^{i-1}, k \cdot 2^i - 2^{i-1} + 1, \ldots, k \cdot 2^i + 2^{i-1} - 1\}$ for $k \in \mathbb{Z}$, on the conclusion of stage 1.
2. $F_i^c(k) = \{k \cdot 2^i, k \cdot 2^i + 1, \ldots, k \cdot 2^i + 2^i - 1\}$ for $k \in \mathbb{Z}$, on the conclusion of stage 2.

We also have two more invariants, namely:

3. $label(a) = k$ for all agents $a \in F_i^c(k)$, for all $i > 0, k \in \mathbb{Z}$.
4. At the end of iteration $i \geq 1$, the agents in group $F_i^c(k)$ are located at position $k \cdot 2^i + \frac{1}{2}(2^i - 1)$ on the line.

Algorithm 1. *Rendezvous on the infinite line*

```
set $\ell = \frac{1}{2}$
foreach agent $a$ do
    set label$(a)$ = position of $a$ on the line
for $i = 1, 2, 3, \ldots$ do
    foreach agent $a$ do
        stage 1:                      /* form the groups $F_i^h(k)$ */
        if odd(label$(a)$) then move right distance $\ell$
            else move left distance $\ell$
        stage 2:                      /* form the groups $F_i^c(k)$ */
        if odd(label$(a)$) then move left distance $2\ell$
            else move right distance $2\ell$
    set $\ell = 2 \cdot \ell$
    set label$(a)$ = $\left\lfloor \frac{\text{label}(a)}{2} \right\rfloor$
```

Theorem 1. *For two agents a_1, a_2 starting at distance d (and at integer points) on the line L, Algorithm 1 permits rendezvous within at most $6d$ synchronized rounds.*

Proof. The four invariants mentioned above are easy to establish by induction.
We first note using the definitions of the sets $F_i^h(k)$ and $F_i^c(k)$ above that for all $i \geq 1$ and $k \in \mathbb{Z}$, we have:

$$F_i^h(k) = F_{i-1}^c(2k-1) \cup F_{i-1}^c(2k),$$
$$F_i^c(k) = F_{i-1}^c(2k) \cup F_{i-1}^c(2k+1).$$

We start with sets $F_1^c(k) = \{2k, 2k+1\}$ for all integers k. Before iteration with index 2, we note that each $F_1^c(k)$ is located at position $2k$.

Inductively, the agents in the group $F_i^c(2k)$ (with label $2k$, by assumption) will first meet those in group $F_i^c(2k-1)$ (with label $2k-1$, again by assumption) in stage 1 of iteration $i+1$, and will then meet those in group $F_i^c(2k+1)$ (with label $2k+1$) in stage 2 of iteration $i+1$. Now, since $label(F_i^c(2k)) = 2k$ (i.e. $label(a) = 2k \ \forall a \in F_i^c(2k)$) and $label(F_i^c(2k+1)) = 2k+1$, we find that all agents in $F_{i+1}^c(k) = F_i^c(2k) \cup F_i^c(2k+1)$ will have label k at the end of iteration $i+1$. Further, by assumption, the group $F_i^c(2k)$ begins iteration $i+1$ at location $2k \cdot 2^i + \frac{1}{2}(2^i - 1)$ on the line. During iteration $i+1$, this group first moves left a distance of 2^{i-1}, then right for a distance of 2^i. Hence, this group ends iteration $i+1$ at location

$$2k \cdot 2^i + \tfrac{1}{2}(2^i - 1) - 2^{i-1} + 2^i = k \cdot 2^{i+1} + \tfrac{1}{2}(2^{i+1} - 1).$$

In a similar manner, we can show that group $F_i^c(2k+1)$ ends iteration $i+1$ at the exact same location as group $F_i^c(2k)$, the pair of them together comprising $F_{i+1}^c(k)$.

Finally, by the end of iteration $i \geq 2$, each agent has met all other agents that began the rendezvous procedure at distance at most 2^{i-1}, and each agent has moved a distance of $3\sum_{j=1}^{i} \ell_j$, where $\ell_j = 2^{j-2}$, as before.

Consider two agents that begin the rendezvous at positions $a_1 < a_2$. Then, let i be the integer such that $2^{i-1} < d = a_2 - a_1 \leq 2^i$. From the claim above, a_1 and a_2 have met by the end of iteration $i+1$. Thus, this process takes time

$$\begin{aligned} 2 + 3\sum_{j=2}^{i+1} \ell_j &= 2 + 3\sum_{j=2}^{i+1} 2^{j-2} = 2 + 3\sum_{j=1}^{i} 2^{j-1} = 2 + 3 \cdot (2^i - 1) \\ &= 6 \cdot 2^{i-1} - 1 < 6d. \end{aligned}$$

We now consider non-integer starting positions of the agents. For a real number $d > 1$, let us define the function $T(d) = 6 \cdot 2^{i-1} - 1$, where i is the integer that satisfies $2^{i-1} < d \leq 2^i$. From Theorem 1, agents starting on integer points at distance d rendezvous in time at most $6d$. This can be carried over to arbitrary (non-integer) starting points and distances, at least in the case when $d \geq 1$.

Lemma 2. *Suppose two agents a_1, a_2 are placed on line L at distance $d \geq 1$. Then, we can adapt Algorithm 1 so that a_1 and a_2 rendezvous within $T(\lceil d \rceil) < 6d$ synchronous rounds.*

Proof. The adaptation of Algorithm 1 is a natural one to consider. An agent not beginning at an integer point initially adopts as its first move a contiguous final segment of the first move of a close by (possibly hypothetical) agent that did begin at an integer point. If the starting location of a is not an integer, then consider $a - \lfloor a \rfloor$ and $\lceil a \rceil - a$. Either one of these two quantities is smaller than the other, or they are equal, i.e. a is closer to one of two integers, or $a = \lfloor a \rfloor + \frac{1}{2}$. In the first case, a adopts the final segment of the first move of the (hypothetical) agent at $\lfloor a \rfloor$ or $\lceil a \rceil$, and then adopts the label of that agent and its behavior for the remaining part of the rendezvous procedure. (If $a = \lfloor a \rfloor + \frac{1}{2}$, we arbitrarily have a adopt the label and procedure of $\lceil a \rceil$.)

Let us assume that $d > 1$. (The case of $d = 1$ is easily handled by a special analysis very similar to the one given below.) Assume that $a_1 < a_2$ and, as before, let $d = a_2 - a_1$. Let us write $d = d^* + \epsilon$, where $d^* = \lfloor d \rfloor$ and $\epsilon < 1$.

There are integers $x < y$ such that $d^* = y - x$

$$x - 1 < a_1 \leq x < y \leq a_2 < y + 1.$$

Since $\epsilon = (x - a_1) + (a_2 - y) < 1$, either $x - a_1$ or $a_2 - y$ is strictly smaller than $\frac{1}{2}$. Assume that $x - a_1 < \frac{1}{2}$. (The other case is similar.)

In this case, by the end of iteration 1, a_1 has adopted the behavior of (the hypothetical) agent x. Similarly, a_2 has adopted the behavior of either agent y or $y+1$ (depending upon a_2's exact location in the interval $[y, y+1)$). This means that a_1 and a_2 will meet in the same time that it takes the agents at positions x and $x + \lfloor d \rfloor$ or x and $x + \lceil d \rceil$ to meet. In particular, this means that a_1 and a_2 will meet in time (at most) $T(\lceil d \rceil)$.

As before, let i be the integer such that $2^{i-1} < d \leq 2^i$. Then we also note that $2^{i-1} < \lceil d \rceil \leq 2^i$. We have noted that a_1 and a_2 will meet in time $T(\lceil d \rceil)$ and since

$$\frac{T(\lceil d \rceil)}{\lceil d \rceil} \leq \frac{T(\lceil d \rceil)}{d} < \frac{6 \cdot 2^{i-1} - 1}{2^{i-1}} < 6,$$

this establishes the bound of the lemma.

In general, it is impossible to define a deterministic rendezvous procedures for agents that start at arbitrarily small distance $d > 0$ and to guarantee that they will meet in time $\mathcal{O}(d)$. In the case where the agents are located very close to each other our algorithm guarantees rendezvous on the conclusion of iteration 2, i.e., in time 4.

3 Linear Time Rendezvous in Trees

We show here that the time complexity of rendezvous in trees is $\mathcal{O}(d)$. We first show that the two agents can meet in time $< 12d$ in the half-line $[0, +\infty)$, where they are allowed to move only in one direction towards the location 0. Later we show how to adopt this algorithm to obtain $\mathcal{O}(d)$ time rendezvous in trees.

3.1 One-Way Rendezvous in the Half-Line

Consider a half-line $L' = [0, +\infty)$ where the agents a_1 and a_2 are labeled by their initial integer positions p_1 and p_2 respectively. Assume also that this time the agents can move only in one direction towards the closed end (0) of L'.

The algorithm is executed in iterations formed of stages 1 and 2. The following invariant is used. During each iteration $i = 2, 3, ...$ the agents are partitioned according to their labels into groups:

1. $F_i^h(k) = \{(k+1)2^i - 2^{i-1} - 1, ..., (k+1)2^i + 2^{i-1} - 2\}$, for $k > 0$, and $F_i^h(0) = \{0, ..., 2^i + 2^{i-1} - 2\}$ on the conclusion of stage 1,
2. $F_i^c(k) = \{(k+1)2^i - 1, ..., (k+2)2^i - 2\}$, for $k > 0$, and $F_i^c(0) = \{0, ..., 2^{i+1} - 2\}$ on the conclusion of stage 2.

Furthermore, we assume that the agents with labels in $F_i^h(k)$ and $F_i^h(k)$ are aligned at position $k \cdot 2^i$ on the conclusion of respective stages. For the completeness of the argument we observe that before the rendezvous algorithm is executed, $F_0^c(k)$ contains the agent with label k located at position $k \geq 0$. Also on the conclusion of iteration 1 each $F_1^c(k)$ contains agents with label $2k+1$ and $2k+2$ located at position $2k$ on the line.

Algorithm 2. *One-way rendezvous*

```
foreach agent $a$ do
    set label($a$) = position of $a$ on the line

for $i = 1, 2, 3, ...$ do
    stage 1:
    form $F_i^h(k) = F_{i-1}^c(2k) \cup F_{i-1}^c(2k+1)$
    by moving agents grouped in $F_{i-1}^c(2k+1)$ by $2^{i-1}$ positions towards 0

    stage 2:
    if $k > 0$ then
        form $F_i^c(k) = F_{i-1}^c(2k+1) \cup F_{i-1}^c(2k+2)$
    else
        form $F_i^c(k) = F_i^h(k) \cup F_{i-1}^c(2k+2)$
    by moving agents grouped in $F_{i-1}^c(2k+2)$ by $2^i$ positions towards 0
```

Corollary 3. *Algorithm 2 has the* enclosure property*, i.e., for any three agents a_1, a_2 and a_3 located initially at positions $0 \leq p_1 < p_2 < p_3$ when agents a_1 and a_3 meet they also meet a_2.*

Proof. The enclosure property is a straightforward consequence of the fact that groups of agents formed on the conclusion of stages 1 and 2 form partitions in which each group is a contiguous segment of positions in L'.

Using reasoning similar to the proof of Theorem 1 we obtain the following.

Theorem 4. *Two agents a_1, a_2 executing Algorithm 2 and initially located at distance d on integer points in the half-line $L' = [0, +\infty)$ require at most $12d$ synchronized rounds to rendezvous.*

3.2 Rendezvous in trees

In this section we assume that the agents a_1 and a_2 are located at some two nodes p_1 and p_2 in a finite tree T. The nodes in the tree are uniquely identified and the agents are aware of the topology of the tree. This allows the agents to select independently a unique node in T that becomes the root r of T. Assume that d_1 and d_2 are respective distances from p_1 and p_2 to r. W.l.o.g., assume that $d_2 \leq d_1$. Let x be the lowest common ancestor (LCA) for p_1 and p_2 in T with respect to the root r, where d_x is the distance separating x from r. For the purpose of rendezvous, the nodes in the tree adopt their distances to r as their labels. For example, the root r adopts the label 0 and node x adopts the label d_x. The new labels d_1 and d_2 of nodes p_1 and p_2 are also adopted by agents a_1 and a_2, respectively. In order to rendezvous, the agents execute Algorithm 2 designed for the half-line $L' = [0, +\infty)$ moving gradually towards the root r with the label 0.

Note that if $x = p_2$, i.e., node p_2 is located on the route from p_1 to r, the distance between p_1 and p_2 is $d = d_1 - d_x$, and according to Theorem 4 the rendezvous process will be completed in time $12d$. Otherwise, the initial distance between p_1 and p_2 in T is $d = d_1 + d_2 - 2d_x$.

Let a node p_2' located on the path from p_1 to r, at distance d_2 from r, be the starting position of a hypothetical agent a_2'. Note that during execution of Algorithm 2 agent a_2' acts the same way as a_2, And in particular the distances between r and a_2 as well as r and a_2' are always the same. Assume also that node x is a starting position of another hypothetical agent a_x.

Due to enclosure property, see Corollary 3, during execution of Algorithm 2 when agents a_x and a_1 meet they also meet a_2'. But since the moves of a_2 and a_2' are identical and all agents move only towards the root r, when agents a_x and a_1 meet, they also meet a_2. Thus according to Theorem 4 agents a_1 and a_2 rendezvous in time $12(d_1 - d_x) < 12(d_1 + d_2 - 2d_x) = 12d$. The following theorem follows.

Theorem 5. *Two agents a_1, a_2 executing Algorithm 2 initially located at distance d on nodes of a rooted tree T can rendezvous in $\leq 12d$ synchronous rounds.*

4 Rendezvous in the Higher-Dimensional Space

In this section we present an algorithm producing the paths of two mobile agents, placed in δ-dimensional grid, which achieve rendezvous in optimal $\mathcal{O}(d)$ time, where d denotes the rectilinear (i.e., ℓ^1) distance between the original positions of the agents. We observe that this result may be used to achieve rendezvous for agents starting at arbitrary initial positions in the δ-dimensional space.

In order to give an efficient synchronous rendezvous algorithm we recall, following [8], the concept of the sequence of *central space partitions* $\Pi = \pi_1, \pi_2, \ldots$. Each π_i is

1. A partition of δ-dimensional space into hypercubes of side length 2^i.
2. The hypercubes are aligned with the axis of the space so they form a δ-dimensional grid.
3. One of these hypercubes is the *central hypercube*, having as its center the origin of the δ-dimensional Cartesian space.

Observe that the corners of hypercubes in π_i are points $u = (u_1, u_2, \ldots, u_\delta)$ such that $\forall r \in \{1, 2, \ldots, \delta\}$, $u_r = 2^{i-1} + k2^i$ for some integer k. Note that all the $(\delta - 1)$-dimensional hyperplanes used in all the partitions of Π are different. 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 incident to v, for $f = 1, 2, \ldots, \delta - 1$.

The idea of the rendezvous algorithm is to direct each agent through a sequence of some potential *meeting points*. These meeting points are the centers of hypercubes of increasing sizes belonging to the successive partitions $\pi_1, \pi_2, \ldots$. The hypercubes are chosen in such a way that the path traversed by each agent is not too long, and eventually both agents reach the same meeting point. Since the agents will traverse, in general, different distances to reach the successive meeting points, their movements will be synchronized with aid of some waiting periods. A couple of lemmas will serve to prove the correctness and the time complexity of our approach.

Lemma 6. *Any hypercube H located in partition π_i intersects the set S of 3^δ hypercubes belonging to partition π_{i-1}. A center of any hypercube from set S is at distance at most $\delta \cdot 2^{i-1}$ from the center of H.*

Proof. We can assume, by symmetry, that H is the central hypercube of partition π_i and s is any of its sides. Observe that the middle point of side s coincides with the center of some hypercube of π_{i-1}. Since the side length of each hypercube of π_i equals 2^i, which is twice the side length of hypercubes of π_{i-1}, s intersects exactly three hypercubes of π_{i-1}. By induction on dimension, all hypercubes of π_{i-1} intersected by H form a hypercube G of side length $3 \cdot 2^i$, i.e. a hypercube whose volume is $3^\delta \cdot 2^{\delta i}$. Hence G is a union of 3^δ hypercubes of the partition π_{i-1}, each one of the volume of $2^{\delta i}$.

To prove the second part of the claim, note that the center of each hypercube of π_{i-1} which intersects H belongs to the closure of H. The shortest rectilinear path from the boundary of H to its center is maximized when the path starts at a vertex of H. Since the length of the side of H equals 2^i, one of its vertices has all δ Cartesian coordinates equal to 2^{i-1}. A mobile agent, moving along the shortest rectilinear path from such vertex to the center of H, in each synchronous round moves from a point x to a point y, such that y has the same coordinates as x, except one coordinate which is reduced by one (with respect to this coordinate for point x). Hence $\delta \cdot 2^{i-1}$ rounds are needed in order to reach the origin (the center of H).

The following lemma can be derived from Lemma 3 in [8].

Lemma 7. *For any pair of points p_1, p_2, initially placed at rectilinear distance d in the δ-dimensional grid, such that $d \leq 2^i$, for some $i = 1, 2, \ldots$ there exists a hypercube H of size at most $2^{i+\delta+1}$ belonging to the hierarchy of partitions Π, such that H contains both points p_1, p_2.*

Now we give an algorithm which achieves rendezvous in linear time of the original distance between the two agents.

Algorithm 3. Rendezvous in the δ-dimensional grid

```
repeat for $i = 1, 2, 3, \ldots$
    set $H$ = hypercube of $\pi_i$ containing your initial position $p$
    move to the center of $H$
    wait until $\delta \cdot 2^{i-1}$ rounds are completed since the start of the current iteration
until rendezvous
```

We prove that the agent's trajectory produced by Algorithm 3 is in $\mathcal{O}(d)$, when d is original distance between the agents, i.e. that Algorithm 3 is optimal (up to a multiplicative constant).

Theorem 8. *Suppose that two location-aware agents are placed at rectilinear distance d in the δ-dimensional grid and the agents simultaneously start their movements produced by Algorithm 3. Then the rendezvous of both agents is achieved within $d \cdot \delta \cdot 2^{\delta+2}$ synchronous rounds.*

Proof. Let i^* be an integer, such that $2^{i^*-1} < d \leq 2^{i^*}$.

Observe that, in the first iteration of the loop of Algorithm 3, the center of the hypercube of side length 2 belonging to π_1 is reached from the initial position of the agent contained within this hypercube in at most δ rounds. By Lemma 6, within the i-th iteration of Algorithm 3, $\delta \cdot 2^{i-1}$ synchronous rounds are sufficient to reach the center of the hypercube H. Therefore the waiting time is sufficiently long, so that at round number $\delta \cdot 2^{i-1}$ of the i-th iteration, both agents are mutually present at the centers of the corresponding hypercubes (despite the fact that the original distance to the center was different for each of them). Hence if two agents aim for the center of the same hypercube at the i-th iteration, they have to meet there before the completion of the iteration. By induction on the iteration number, both of the agents are synchronized so they start and finish their movement of each subsequent iteration at the same moments of time. By Lemma 7, both of the agents eventually aim for the center of the same hypercube H, whose side length equals at most $2^{i^*+\delta+1}$, and they meet there. Such hypercube H belongs to partition $\pi_{i^*+\delta+1}$, hence its center is reached by the agents in iteration $i^* + \delta + 1$. The number of rounds spent by each agent until the end of iteration $i^* + \delta + 1$ equals

$$\sum_{1 \leq i \leq i^*+\delta+1} \delta \cdot 2^{i-1} < \delta \cdot 2^{i^*+\delta+1} < d \cdot \delta \cdot 2^{\delta+2} \quad .$$

Algorithm 3 may be adapted to achieve rendezvous in δ-dimensional space. It is sufficient that the agents construct a common grid, each agent moves first to the closest grid position and then they continue their movements according to Algorithm 3. The cost of such algorithm becomes $d \cdot \delta \cdot 2^{\delta+2} + \delta$, where δ extra rounds are used first by each agent to reach the closest grid position.

Corollary 9. *Suppose that two location-aware agents are placed at rectilinear distance d in the δ-dimensional space. The rendezvous of the agents may be achieved within $\delta(d \cdot 2^{\delta+2} + 1)$ synchronous rounds.*

For the Euclidean distance case, when the agents do not need to walk along the axis of the δ-dimensional space, a finer grid may be constructed, so each agent may reach a grid point in a single step and the time given by Corollary 9 reduces to $\delta \cdot d \cdot 2^{\delta+2} + 1$. For this purpose it is sufficient to scale down the grid by at least a factor of $\frac{\sqrt{\delta}}{2}$.

5 Rendezvous in Arbitrary Graphs

In this section we assume that the agents a_1 and a_2 are located at some two nodes of a finite graph. In the same discrete model as for the earlier-studied case of trees, the agents are aware of the topology of the graph and all nodes are uniquely identified. We start by showing that, unlike in all the cases discussed so far, there exist graph instances which require $\Omega(d)$ time to achieve rendezvous. In fact, rendezvous time of $\Omega(d\frac{\log n}{\log\log n})$ is required for graphs of extremal girth, with logarithmic average degree. In the following, all logarithms are assumed to be base 2.

Theorem 10. *Let ϵ be arbitrarily fixed. For any integers $N > 8$ and $d > 0$, such that $d < \min\{N^{1-\epsilon}, N/8\}$, there exists a graph of order N, such that for any rendezvous algorithm $\mathcal{A}$ there is a pair of starting locations at distance d such that rendezvous using algorithm $\mathcal{A}$ requires at least $\frac{\epsilon}{4}\frac{d\log N}{\log\log N}$ rounds, even when assuming simultaneous start.*

Proof. From ([11], Theorem III.1.1) it follows that for any integer $k > 3$, there exists a graph $G = (V, E)$ with $n = 2^k$ vertices, having $m = \frac{1}{2}nk$ edges, whose shortest cycle is of length $g > k/\log k + 1$. Let $\mathcal{A}$ be any algorithm defining the behavior of an agent. We will first show that there exists a pair of neighboring vertices in G such that the time required for rendezvous of agents starting from this pair of vertices (with $d = 1$), using algorithm $\mathcal{A}$, is at least equal to $\min\{(g-1)/2, k/2\}$. Suppose, to the contrary, that for all possible initial locations of the agents in G, the agents rendezvous within some time $t < \min\{(g-1)/2, k/2\}$. For all $v \in V$, let $A_v \subseteq G$ be the subgraph spanned by the edges visited by an agent following algorithm $\mathcal{A}$, starting from vertex v, during the first t rounds, under the assumption that the agent does not meet the other agent. Each graph A_v has at most t edges. For any pair of adjacent vertices u, v of G, the intersection of graphs A_u and A_v must contain at least one vertex w; otherwise, no vertex will be visited by both of the agents starting at u and v, hence such agents cannot meet. Both of the graphs A_u and A_v are connected and of diameter at most $t < (g-1)/2$. Consequently, we must have $w \in \{u, v\}$, since otherwise G would contain a cycle of length less than g, obtained by traversing the shortest path from v to w in A_v, traversing the shortest path from w to u in A_u, and finally traversing the edge $\{u, v\}$. It follows that u is a vertex of A_v, or v is a vertex of A_u; without loss of generality, assume the former case. Then, the edge $\{u, v\}$ belongs to A_v, since otherwise the shortest path connecting u and v in A_v would form a cycle with edge $\{u, v\}$ of length less than g. Since u and v were arbitrarily chosen, it follows that each edge $e \in E$ belongs to some graph A_v, i.e., $e \in \bigcup_{v\in V} E(A_v)$. Consequently, $\sum_{v\in V} |E(A_v)| \geq m$, and so, there must exist a vertex v such that $|E(A_v)| \geq m/n \geq k/2$. However, this is a contradiction with $|E(A_v)| \leq t < k/2$. It follows that $t \geq \min\{(g-1)/2, k/2\} \geq k/(2\log k)$. Given a value of N, we consider an input graph on N vertices consisting of graph G for $k = \lfloor \log N \rfloor$ (with $n = 2^k$), and $N - n$ additional vertices, attached with single edges to an arbitrarily chosen vertex of G. Since this modification does not affect rendezvous time, we obtain for any value of N a lower-bound of $\frac{1}{2}\frac{\lfloor \log N \rfloor}{\log\lfloor \log N \rfloor}$ on rendezvous time for agents starting from neighboring vertices, i.e., for $d = 1$.

To prove the claim of the theorem for larger values of d, we construct a graph $G' = (V', E')$ from $G = (V, E)$ by inserting a path of $d - 1$ vertices on each edge of G. Graph G' has $n' = n + m(d-1)$ vertices and $m' = md$ edges. Let $\mathcal{A}$ be a fixed algorithm for an agent which always reaches rendezvous in G' in time at most $t < d\min\{(g-1)/2, k/2\}$, and let $A'_v \subseteq G'$, for $v \in V$, be defined as the subgraphs spanned by the edges visited by an agent following algorithm $\mathcal{A}$, starting from vertex v during the first t rounds, under the assumption that the agent does not meet the other agent. By an argument similar to that for the case of $d = 1$, we have that for all $e' \in E'$, $e' \in \bigcup_{v \in V} E(A'_v)$, and consequently, there exists a vertex $v \in V$ such that $|E(A'_v)| \geq m'/n = dm/n \geq dk/2$. This is a contradiction with $|E(A'_v)| \leq t < dk/2$. In this way we obtain a lower-bound of $dk/(2 \log k)$ on rendezvous time in G'. Given a value of N, we consider an input graph on N vertices consisting of graph G' for $k = \lfloor \log(N/d) \rfloor$ (with $n' = d2^k$), and $N - n'$ additional vertices, attached with single edges to an arbitrarily chosen vertex of G'. Since this modification does not affect rendezvous time, we obtain for any value of N a lower-bound of $\frac{1}{2} \frac{d \lfloor \log(N/d) \rfloor}{\log \lfloor \log(N/d) \rfloor} > \frac{\epsilon}{4} \frac{d \log N}{\log \log N}$ on rendezvous time for agents starting at distance d, where we have taken into account that $d < N^{1-\epsilon}$.

In conclusion, we point out that a poly-logarithmic overhead in n is sufficient to achieve rendezvous, assuming simultaneous start of the agents.

Theorem 11. *Suppose that two location-aware agents are placed at a distance of d in a known arbitrary graph $G = (V, E)$ of order n. Then agents with simultaneous start can rendezvous within $\mathcal{O}(d \log^2 n)$ synchronous rounds.*

Theorem 12. *Suppose that two location-aware agents are placed at a distance of d in a known graph $G = (V, E)$ of order n. Then the rendezvous of agents with simultaneous start is achieved within $\mathcal{O}(d)$ synchronous rounds if G is a circular-arc graph, and within $\mathcal{O}(d \log n)$ synchronous rounds if G is a chordal graph or a cocomparability graph.*

References

1. Alpern, S.: The rendezvous search problem. SIAM Journal on Control and Optimization 33, 673–683 (1995)
2. Alpern, J., Baston, V., Essegaier, S.: Rendezvous search on a graph. Journal of Applied Probability 36, 223–231 (1999)
3. Alpern, S.: Rendezvous Search: A Personal Perspective. Operations Research 50(5), 772–795 (2002)
4. Alpern, S., Gal, S.: The Theory of Search Games and Rendezvous. Kluwer Academic Publishers, Dordrecht (2003)
5. Anderson, E., Fekete, S.: Asymmetric rendezvous on the plane. In: Proc. 14th Annual ACM Symposium on Computational Geometry, pp. 365–373 (1998)
6. Anderson, E., Fekete, S.: Two-dimensional rendezvous search. Operations Research 49, 107–118 (2001)
7. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Transactions on Robotics and Automation 15(5), 818–828 (1999)
8. Bampas, E., Czyzowicz, J., Gąsieniec, L., Ilcinkas, D., Labourel, A.: Almost Optimal Asynchronous Rendezvous in Infinite Multidimensional Grids. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 297–311. Springer, Heidelberg (2010)
9. Baston, V., Gal, S.: Rendezvous on the line when the player's initial distance is given by an unknown probability distribution. SIAM J. on Control and Optimization 36, 1880–1889 (1998)

10. Baston, V., Gal, S.: Rendezvous search when marks are left at the starting points. Naval Research Logistics 48, 722–731 (2001)
11. Bollobas, B.: Extremal Graph Theory. Academic Press, London (1978)
12. Cohen, R., Peleg, D.: Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. SIAM Journal on Computing 34(6), 1516–1528 (2005)
13. Collins, A., Czyzowicz, J., Gąsieniec, L., Labourel, A.: Tell me where I am so I can meet you sooner (Asynchronous rendezvous with location information). In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 502–514. Springer, Heidelberg (2010)
14. Czyzowicz, J., Kosowski, A., Pelc, A.: How to meet when you forget: Log-space rendezvous in arbitrary graphs. In: Proc. 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2010), pp. 450–459 (2010)
15. Czyzowicz, J., Labourel, A., Pelc, A.: How to Meet Asynchronously (Almost) Everywhere. In: Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (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. Dragano, F., Yano, C., Lomonosov, I.: Collective tree spanners of graphs. SIAM Journal on Discrete Mathematics 20, 240–260 (2006)
19. Emek, Y., Gąsieniec, 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. 27th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2008), pp. 223–232 (2008)
21. Gaber, I., Mansour, Y.: Broadcast in radio networks. In: Proc. 6th ACM Symposium on Discrete Algorithms (SODA 1995), pp. 577–585 (1995); Also in J. of Algorithms 46, 1–20 (2003)
22. Gal, S.: Rendezvous search on the line. Operations Research 47, 974–976 (1999)
23. Kowalski, D., Malinowski, A.: How to meet in anonymous network. Theoretical Computer Science 399, 141–156 (2008)
24. Kozma, G., Lotker, Z., Sharir, M., Stupp, G.: Geometrically aware communication in random wireless networks. In: Proc. 23rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004), pp. 310–319 (2004)
25. Kranakis, E., Krizanc, D., Markou, E.: The Mobile Agent Rendezvous Problem in the Ring. Lectures on Distributed Computing Theory. Morgan and Claypool Publishers (2010)
26. Kranakis, E., Krizanc, D., Rajsbaum, S.: Mobile agent rendezvous: A survey. In: Flocchini, P., Gąsieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 1–9. Springer, Heidelberg (2006)
27. Olfati-Saber, R., Fax, J.A., Murray, R.M.: Consensus and Cooperation in Networked Multi-Agent Systems. Proc. of IEEE 95(1), 215–233 (2007)
28. Souissi, S., Défago, X., Yamashita, M.: Using eventually consistent compasses to gather memory-less mobile robots with limited visibility. ACM Transactions on Autonomous and Adaptive Systems 4(1) (2009)
29. Suzuki, I., Yamashita, M.: Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. SIAM Journal on Computing 28(4), 1347–1363 (1999)
30. 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)

Toward a Formal Semantic Framework for Deterministic Parallel Programming*

Li Lu and Michael L. Scott

Computer Science Department, University of Rochester
Rochester, NY 14627-0226 USA
{llu,scott}@cs.rochester.edu

Abstract. Deterministic parallelism has become an increasingly attractive concept: a deterministic parallel program may be easier to construct, debug, understand, and maintain. However, there exist many different definitions of "determinism" for parallel programming. Many existing definitions have not yet been fully formalized, and the relationships among these definitions are still unclear. We argue that formalism is needed, and that history-based semantics—as used, for example, to define the Java and C++ memory models—provides a useful lens through which to view the notion of determinism. As a first step, we suggest several history-based definitions of determinism. We discuss some of their comparative advantages, prove containment relationships among them, and identify programming idioms that ensure them. We also propose directions for future work.

1 Introduction

Determinism, loosely defined, is increasingly touted as a way to simplify the design, verification, testing, and debugging of parallel programs—in effect, as a way to make it easier to understand what a parallel program does. Parallel functional languages have long enjoyed the benefits of determinism [18]. Recent workshops have brought together members of the architecture, programming languages, and systems communities to discuss determinism in more general languages and systems [13,1]. Determinism has also featured prominently in recent workshops on pedagogy for concurrency [23,28,30].

At the very least, determinism suggests that a given parallel program—like a sequential program under most semantic models—should always produce the same output when run with the same input. We believe, however, that it needs to mean more than this—that runs of a deterministic program on a given input should not only produce the same output: they should produce it *in the same way*. By analogy to automata theory, a deterministic Turing machine doesn't just compute a single-valued function: it takes a uniquely determined action at every step along the way.

For real-world parallel programs, computing "in the same way" may be defined in many ways. Depending on context, we may expect that repeated runs of a deterministic

* This work was supported in part by the US National Science Foundation under grant CCR-0963759.

program will consume (more or less) the same amount of time and space; that they will display the same observable intermediate states to a debugger; that the behavior of distributed replicas will not diverge; that the number of code paths requiring separate testing will be linear in the number of threads (rather than exponential); or that the programmer will be able to straightforwardly predict the impact of source code changes on output or on time and space consumption.

History-based semantics has proven to be one of the most useful ways to model the behavior of parallel programs. Among other things, it has been used to explain the serializability of transactions [27], the linearizability of concurrent data structures [19], and the memory model that determines the values seen by reads in a language like Java [22] or C++ [11]. Memory models typically distinguish between *ordinary* and *synchronizing* accesses, and use these to build a cross-thread partial order among operations of the program as a whole. Recently we have proposed that the various sorts of synchronizing accesses be unified under the single notion of an *atomic action* [29, 15].

Informally, the parallel semantics of a given program on a given input is a set of *abstract executions*. Each execution comprises a set of *thread histories*, each of which in turn 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 language's (separately specified, typically operational) sequential semantics, the program's input, and the values returned by reads (which may have been set by writes in other threads). An execution is said to be *sequentially consistent* if there exists a total order on reads and writes, consistent with program order in every thread, such that each read returns the value written by the most recent preceding write to the same location. Under relaxed memory models, a program is said to be *data-race free* if the model's partial order covers all pairs of conflicting operations.

An *implementation* maps source programs to sets of low-level *target executions* on some real or virtual machine. The implementation is correct only if, for every target execution, there exists a corresponding abstract execution that performs the same external actions, in the same order. (The extent to which shorter sequences of target-level operations must correspond to operations of the abstract execution is related to, but separate from, the subject of this paper; we do not address it further here.)

In the strictest sense of the word, a deterministic parallel program would be one whose semantics, on any given input, consists of only a single abstract execution, to which any legal target execution would have to correspond. In practice, this definition may prove too restrictive. Suppose, for example, that I have chosen, as a programmer, to "roll my own" shared allocator for objects of some heavily used data type, and that I am willing to ignore the possibility of running out of memory. Suppose further that my allocator keeps free blocks on a simple lock-free stack. Because they access a common top-of-stack pointer, allocation and deallocation operations must synchronize with one another, and will thus be ordered in any given execution. Since I presumably don't care what the order is, I may wish to allow arbitrary executions that differ only in the order realized, while still saying that my program is deterministic.

In general, we suggest, it makes sense to say that a program is deterministic if all of its abstract executions on a given input are *equivalent* in some well-defined sense. A *language* may be said to be deterministic if all its programs are deterministic. An *implementation* may be said to be deterministic (for a given, not-necessarily-deterministic language) if all the target executions of a given program on a given input correspond to abstract executions that are mutually equivalent. For all these purposes, ***the definition of determinism amounts to an equivalence relation on abstract executions.***

We contend that history-based semantics provides a valuable lens through which to view determinism. By specifying semantics in terms of executions, we capture the notion of "computing in the same way"—not just computing the same result. We also accommodate programs (e.g., servers) that are not intended to terminate—executions need not be finite. By separating semantics (source-to-abstract-execution) from implementation (source-to-target-execution), we fix the level of abstraction at which determinism is expected, and, with an appropriate definition of "equivalence," we codify what determinism *means* at that level.

For examples like the memory allocator mentioned above, history-based semantics highlights the importance of language definition. If my favorite memory management mechanism were a built-in facility, with no implied ordering among allocation and deallocation operations of different objects, then a program containing uses of that facility might still have a single abstract execution. Other potential sources of nondeterminism that might be hidden inside the language definition include parallel iterators, bag-of-task work queues, and container data types (sets, bags, mappings). Whether *all* such sources can reasonably be shifted from semantics to implementation remains an open question (but we doubt it).

In a similar vein, an implementation may be deterministic for only a subset of some standard programming language—i.e., for a smaller language. MIT's Kendo system, for example, provides determinism for only those $\langle$program, input$\rangle$ pairs that are data-race free—a property the authors call *weak determinism* [26].

From an implementation perspective, history-based semantics differentiates between things that are required to be deterministic and things that an implementation might *choose* to make deterministic. This perspective draws a sharp distinction between projects like DPJ [10], Prometheus [3], and CnC [12], which can be seen as constraining the set of abstract executions, and projects like Kendo, DMP [16], CoreDet [5], and Grace [8], which can provide deterministic execution even for (some) pthread-ed programs in C. (Additional projects, such as Rerun [20], DeLorean [24], and DoublePlay [31], are intended to provide deterministic *replay* of a program whose initial run is more arbitrary.)

If we assume that an implementation is correct, history-based semantics identifies the set of executions that an application-level test harness might aspire to cover. For purposes of debugging, it also bounds the set of global states that might be visible at a breakpoint—namely, those that correspond to a consistent cut through the partial order of a legal abstract execution.

We believe the pursuit of deterministic parallel programming will benefit from careful formalization in history-based semantics. Toward that end, we present a basic system model in Section 2, followed by several possible definitions of equivalence for abstract executions in Section 3. We discuss the comparative advantages of these definitions in Section 4. We also prove containment relationships among the definitions, and identify programming idioms that ensure them. Many other definitions of equivalence are possible, and many additional questions seem worth pursuing in future work; we list a few of these in Section 5.

2 System Model

In a manner consistent with standard practice and with our recent work on atomicity-based semantics [15], we define an *execution* of a program P, written in a language $\mathcal{L}$, to be a 3-tuple $E_{P,\mathcal{L}} : (OP, <_p, <_s)$, where *OP* is a set of *operations*, and $<_p$ (*program order*) and $<_s$ (*synchronization order*) are irreflexive partial orders on *OP*. When we can do so without confusion, we omit P and $\mathcal{L}$ from our notation.

Each operation in *OP* takes one of six forms: (read, name, val, tid, uid), (write, name, val, tid, uid), (input, val, tid, uid), (output, val, tid, uid), (begin_atomic, tid, uid), or (end_atomic, tid, uid). In each of these, tid identifies the executing thread. Uid is an arbitrary unique identifier; it serves to make every operation distinct and to allow the set *OP* to contain multiple operations that are otherwise identical. In read and write operations, name identifies a program variable; in read, write, input, and output operations, val identifies a value read from a variable or from the program's input, or written to a variable or to the program's output. The domains from which thread ids, variable names, and values are chosen are defined by the semantics of $\mathcal{L}$. These domains are assumed to be countable, but not necessarily finite.

Program order, $<_p$, is a union of disjoint total orders, one per thread. Specifically, if o_1 = $(\ldots, t_1, u_1)$ and o_2 = $(\ldots, t_2, u_2)$ are distinct operations of the same execution (i.e., $u_1 \neq u_2$), then $(t_1 = t_2) \rightarrow (o_1 <_p o_2 \veebar o_2 <_p o_1)$ and $(t_1 \neq t_2) \rightarrow (o_1 \not<_p o_2 \wedge o_2 \not<_p o_1)$. (Here $\veebar$ is *exclusive or*.)

For any given thread t_i, $OP|_{t_i}$ or $E|_{t_i}$ represents t_i's *thread history*—its (totally ordered) sequence of operations. We use $OP|_s$ or $E|_s$ to represent an execution's *synchronization operations*: begin_atomic, end_atomic, input, and output. We use $OP|_e$ or $E|_e$ to represent the execution's *external operations*: input and output. Clearly $OP|_e \subset OP|_s$. (We assume that I/O races are always unacceptable.)

Synchronization order, $<_s$, is a total order on $OP|_s$. It does not relate reads and writes, but it is consistent with program order. That is, for o_1 = $(\ldots, t_1, u_1)$ and o_2 = $(\ldots, t_2, u_2)$, $(o_1 <_s o_2 \wedge t_1 = t_2) \rightarrow (o_1 <_p o_2)$.

For convenience, we define v_{in} and v_{out}, for a given execution E, to be the execution's *input* and *output vectors*—the (possibly infinite) sequences of values contained, in order of $<_s$, in E's input and output operations, respectively. For any given execution E, we use $ext(E)$ to represent the pair $\langle v_{in}, v_{out}\rangle$.

We use begin_atomic and end_atomic operations in our model to capture the synchronization operations of $\mathcal{L}$, whatever those may be—thread fork and join, lock acquire and release, monitor entry and exit, volatile variable read and write, etc. For this reason, we require that begin_atomic and end_atomic operations appear in disjoint, unnested pairs, and never bracket input or output operations. That is, for every b = (begin_atomic, t, u_1) there exists an e = (end_atomic, t, u_2) such that $b <_s e$ and $\forall m \in OP|_s \setminus \{b, e\},\ m <_s b \vee e <_s m$; likewise for every e = (end_atomic, t, u_2) there exists a b = (begin_atomic, t, u_1) such that $b <_s e$ and $\forall m \in OP|_s \setminus \{b, e\},\ m <_s b \vee e <_s m$. We use $OP|_a$ or

$E|_a$ to represent the execution's *atomic actions*: the union of $E|_e$ and the set of minimal sequences of operations in each thread beginning with begin_atomic and ending with end_atomic.

Continuing with standard practice, we assume that the semantics of $\mathcal{L}$ defines, for any given execution, a *synchronizes-with order*, $<_{sw}$, that is a subset of $<_s$—that is, a partial order on $OP|_s$. In a lock-based language, for example, the *release* method for lock L might be modeled as (begin_atomic, t_1, u_1), (write, L, 0, t_1, u_2), (end_atomic, t_1, u_3);[1] an *acquire* might be (begin_atomic, t_2, u_4), (read, L, 0, t_2, u_5), (write, L, 1, t_2, u_6), (end_atomic, t_2, u_7). If operation u_3 precedes operation u_4 in $<_s$, we might require that it do so in $<_{sw}$ as well (since they operate on the same lock), but operations used to model methods of different locks might be unrelated by $<_{sw}$.

Given $<_{sw}$, we define *happens-before order*, $<_{hb}$, to be the irreflexive transitive closure of $<_p$ and $<_{sw}$. Finally, we assume that $\mathcal{L}$ defines, given $<_{hb}$, a *reads-see-writes* function W that specifies, for any given read operation r, the set of write operations $\{w_i\}$ whose values r is permitted to return. In most languages, r will be allowed to see w if w is the most recent previous write to the same variable on some happens-before path. In some languages (e.g., Java), r may be allowed to see w if the two operations are incomparable under $<_{hb}$. Languages may also differ as to whether atomic actions are *strongly atomic*—that is, whether nonatomic reads can see inside them, or nonatomic writes be seen within them [9] [15, TR version appendix].

In any consistent cut across $<_{hb}$, we define the *program state* to be (1) the prefixes of v_{in} and v_{out} that have been input and output prior to the cut, and (2) the most recent values written to the program's variables according to $<_{hb}$. If a variable has not yet been written, its value is undefined ($\perp$); in a program with a data race, the most recent write may not be unique, in which case the variable's value is indeterminate.

Two operations (read or write) *conflict* if they access the same variable and at least one of them writes it. An execution is *data-race free* if all conflicting operations are ordered by $<_{hb}$.

In this paper, we consider only *well-formed* executions. An execution E is well formed if and only if it satisfies the following three requirements.

Adherence to per-thread semantics: Given the code for thread t and the values returned by read and input operations, $\mathcal{L}$'s (independently specified) sequential semantics determine the set of legal histories for t; $E|_t$ must be among them. Moreover, begin_atomic and end_atomic operations in $E|_t$ must occur in disjoint matched pairs, with only read or write operations between them (as ordered by $<_p$).

Consistent ordering: For all t, operations of $E|_t$ are totally ordered by $<_p$. Operations with different tids are unordered by $<_p$. $E|_s$ is totally ordered by $<_s$, which is consistent with $<_p$. Reads and writes do not participate in $<_s$. Paired begin_atomic and end_atomic operations are contiguous in $<_s$.

Adherence to memory model: All values read are permitted by W, the reads-see-writes function induced by $<_p$, $<_s$, $<_{sw}$, and $<_{hb}$, according to $\mathcal{L}$'s semantics.

[1] The release sequence must be bracketed with begin_atomic. . . end_atomic, even though there is only one operation inside, in order to induce cross-thread ordering.

3 Example Definitions of Equivalence

In this section we suggest several possible definitions of equivalence for abstract executions. Two—*Singleton* and *ExternalEvents*—are intended to be extreme cases: the strictest and loosest definitions that strike us as plausible. Another—*FinalState*—is similar to *ExternalEvents*, restricted to programs that terminate. The other two—*Dataflow* and *SyncOrder*—are two of many possible in-between options.

Singleton. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if they differ only in the* uid*s of their operations; that is, there exists a one-one mapping (bijection) between OP_1 and OP_2 that preserves $<_p$, $<_s$, and the content other than* uid *in every operation.*

Singleton uses the strictest possible definition of determinism: there must be only one possible execution for a given program and input.

Dataflow. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if $ext(E_1) = ext(E_2)$ and there is a one-one mapping between OP_1 and OP_2 that preserves (1) the content other than* tid *and* uid *in every operation, and (2) the reads-sees-writes function W induced, under $\mathcal{L}$'s semantics, by $<_p$, $<_s$, $<_{sw}$, and $<_{hb}$.*

Informally, *Dataflow* requires that reads see the same writes in both executions, and that the values in both reads and writes (including the input and output operations that "read" and "write" elements of v_{in} and v_{out}) be the same in both executions. Note that we do not require that the bijection preserve $<_p$ or $<_s$, nor do we require that the executions be data-race free.

SyncOrder. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if there is a one-one mapping between $E_1|_a$ and $E_2|_a$ that preserves (1) $<_s$, and (2) the content other than* uid *in every synchronization operation and in every* read *or* write *within an atomic action.*

SyncOrder requires that there be a fixed pattern of synchronization among threads in E_1 and E_2, with atomic actions reading and writing the same values in the same variables. Note that if executions are data-race free (something that *SyncOrder* does not require), then they are also sequentially consistent [2], so $E_1 \equiv_{\text{SyncOrder}} E_2 \wedge E_1, E_2 \in DRF \rightarrow E_1 \equiv_{\text{Dataflow}} E_2$.

ExternalEvents. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if $ext(E_1) = ext(E_2)$.*

ExternalEvents is the most widely accepted language-level definition of determinism. It guarantees that abstract executions on the same input look "the same" from the perspective of the outside world.

FinalState. *Executions $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are said to be equivalent if and only if they both terminate and their program states at termination (values of variables and of v_{in} and v_{out}) are the same.*

Like *ExternalEvents*, *FinalState* says nothing about how E_1 and E_2 compute. It requires only that final values be the same. Unlike *ExternalEvents*, *FinalState* requires agreement on variables *other* than output.

4 Discussion

Singleton is the strictest definition of equivalence, and thus of determinism. It is a common notion in the literature—corresponding, for example, to what Emrath and Padua called "internally determinate" [17] and Netzer and Miller "internally deterministic" [25]. It requires a single execution for any given source program and input. Interestingly, while we have not insisted that such executions be sequentially consistent, they seem likely to be so in practice: a language that admits non-sequentially consistent executions (e.g., via data races) seems likely (unless it is designed in some highly artificial way) to admit multiple executions for some ⟨program, input⟩ pairs.

By requiring abstract executions to be identical in every detail, *Singleton* rules out "benign" differences of any kind. It may therefore preclude a variety of language features and programming idioms that users might still like to think of as "deterministic."

Dataflow relaxes *Singleton* by loosening the requirements on control flow. Equivalent executions must still have the same operation sets (ignoring tid and uid), but the synchronization and program orders can be different, so long as values flow from the same writes to the same reads. In the literature, *Dataflow* is essentially equivalent to Karp and Miller's 1966 definition of "determinacy" [21], which was based on a dataflow model of computation. Intuitively, *Dataflow* can be thought of as an attempt to accommodate programming languages and idioms in which the work of the program is fixed from run to run, but may be partitioned and allocated differently among the program's threads.

SyncOrder also relaxes *Singleton*, but by admitting benign changes in data flow, rather than control flow. Specifically, *SyncOrder* requires equivalent executions to contain the exact same synchronization operations, executed by the same threads in the same order. It does *not* require that a read see the same write in both executions, but it does require that any disagreement have no effect on synchronization order (including output).

ExternalEvents is also a common notion in the literature. It corresponds to what Emrath and Padua called "externally determinate" [17] and, more recently, to the working definition of determinism adopted by Bocchino et al. [10]. The appeal of the definition lies in its generality. If output is all one cares about, *ExternalEvents* affords the language designer and implementor maximum flexibility. From a practical perspective, knowing that a parallel program will always generate the same output from the same input, regardless of scheduling idiosyncrasies, is a major step forward from the status quo. For users with a strong interest in predictable performance and resource usage, debugability, and maintainability, however, *ExternalEvents* may not be enough.

FinalState is essentially a variant of *ExternalEvents* restricted to programs that terminate (and whose internal variables end up with the same values in every execution). It corresponds to what Netzer and Miller called "externally deterministic" [25].

In the remainder of this section, we explore additional ramifications of our example definitions. In Section 4.1 we formalize containment properties: which definitions of equivalence, if they hold between a given pair of executions, imply which other definitions? Which definitions are incomparable? In Section 4.2 we identify programming languages and idioms that illustrate these containments. Finally, in Sections 4.3 and 4.4, we consider the practical issues of repetitive debugging and of deterministic implementation for nondeterministic languages.

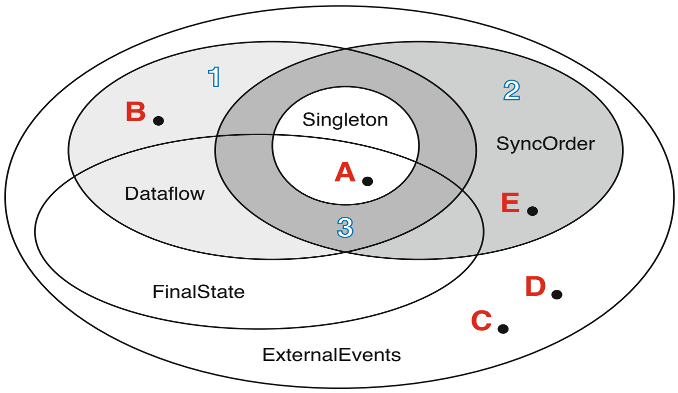


Fig. 1. Containment relationships among definitions of determinism, or, equivalently, abstract execution equivalence. Names of equivalence definitions correspond to ovals. Outlined numbers label the light, medium, and dark shaded regions. Bold letters show the locations of programming idioms discussed in Section 4.2.

4.1 Containment Properties

Figure 1 posits containment relationships among the definitions of determinism given in Section 3. The space as a whole is populated by sets $\{X_i\}$ of executions of some given program on a given input, with some given semantics. If region S is contained in region L, then all executions that are equivalent under definition S are equivalent under definition L as well; that is, S is a stricter and L a looser definition. (The regions can also be thought of as containing languages or executions: a language [execution] is in region R if for every program and input, all abstract executions generated by the language semantics [or corresponding to target executions generated by the implementation] are equivalent under definition R.) We justify the illustrated relationships as follows.

Theorem 1. Singleton *is contained in* Dataflow, SyncOrder, *and* ExternalEvents.

Proof: Suppose E_1 and E_2 are arbitrary equivalent executions under *Singleton*. By definition, E_1 and E_2 are identical in every respect other than the uids of their operations. None of the definitions of *Dataflow*, *SyncOrder*, or *ExternalEvents* speaks to uids. Each requires certain other portions of E_1 and E_2 (or entities derived from them) to be the same; *Singleton* trivially ensures this. □

Theorem 2. Singleton, Dataflow, SyncOrder, *and* FinalState *are all contained in* ExternalEvents.

Proof: For *Singleton*, this is proved in Theorem 1. For *Dataflow*, it follows from the definition: if E_1 and E_2 are *Dataflow* equivalent, then $ext(E_1) = ext(E_2)$.

For *SyncOrder*, suppose $E_1 : (OP_1, <_{p1}, <_{s1})$ and $E_2 : (OP_2, <_{p2}, <_{s2})$ are arbitrary equivalent executions under *SyncOrder*. This means there is a bijection between OP_1 and OP_2 that preserves (among other things) both $<_s$ and the content other than uid in each input and output operation. Since input and output operations are totally ordered by $<_s$, and since v_{in} and v_{out} are defined to be the values in an execution's input and output operations, in order of $<_s$, we have $ext(E_1) = ext(E_2)$.

For *FinalState*, suppose E_1 and E_2 are equivalent under *FinalState*. Then E_1 and E_2 both terminate, and with the same state. Their input and output vectors, included in their terminating states, must therefore be the same: $\mathit{ext}(E_1) = \mathit{ext}(E_2)$. □

Theorem 3. *There are sets of executions that are equivalent under* Dataflow *but not under* SyncOrder.

Proof: This is the light gray region, labeled "1" in Figure 1. It corresponds to programs with benign synchronization races. Consider a program in which two threads each increment a variable under protection of a lock: acquire(L); x++; release(L). Under plausible semantics, one possible execution looks as follows (ignoring uids), where $<_p$ orders the operations of each thread as shown, and $<_s$ orders the atomic actions of thread 1 before those of thread 2:

(begin_atomic, t_1) (read, L, 0, t_1) (write, L, 1, t_1) (end_atomic, t_1)
    (read, x, 0, t_1) (write, x, 1, t_1) (begin_atomic, t_1) (write, L, 0, t_1) (end_atomic t_1)
(begin_atomic, t_2) (read, L, 0, t_2) (write, L, 1, t_2) (end_atomic, t_2)
    (read, x, 1, t_2) (write, x, 2, t_2) (begin_atomic, t_2) (write, L, 0, t_2) (end_atomic t_2)

Call this execution E_1. Another execution (call it E_2) looks the same, except that the tids in various operations are reversed: thread 2 changes x from 0 to 1; thread 1 changes it from 1 to 2. For *Dataflow*, the obvious bijection swaps the tids, and the executions are equivalent. For *SyncOrder*, there is clearly no bijection that preserves both synchronization order and the tids in each begin_atomic and end_atomic operation. □

Theorem 4. *There are sets of executions that are equivalent under* SyncOrder *but not under* Dataflow.

```
        Initially flag == 0

t1:                 t2:
   flag = 1            flag = 2
                       if (flag > 0)
                             print "flag > 0"
                             print "end"
```

Proof: This is the medium gray region, labeled "2" in Figure 1. It corresponds to programs with benign data races. An example is shown at right. This is a racy program: neither the write nor the read of flag in t2 is ordered by $<_{hb}$ with the write in t1. Two abstract executions, E_1 and E_2 (not shown), may thus have different data flow: in E_1, the read of flag in t2 returns the value 1, while in E_2 it returns the value 2. However, these two executions have the same synchronization order: in both, only the two outputs are ordered by $<_s$, and they are ordered the same in both executions. Thus $E_1 \equiv_{\mathit{SyncOrder}} E2$ but $E_1 \not\equiv_{\mathit{Dataflow}} E_2$. □

Theorem 5. Singleton*,* Dataflow*, and* SyncOrder *all have nontrivial intersections with* FinalState.

Proof: *Singleton*, *Dataflow*, and *SyncOrder* clearly all contain sets of terminating executions that have the same final state. However, equivalent executions in *Singleton*, *Dataflow* or *SyncOrder* do not necessarily terminate. Suppose E_1 and E_2 are arbitrary equivalent executions under *Singleton*, *Dataflow* or *SyncOrder*, and also under *FinalState*. We can make both executions nonterminating by adding an additional thread to each that executes an infinite but harmless loop (it might, for example, read an otherwise unused variable over and over). The modified executions will no longer be in *FinalState* since they do not terminate, but they will still be in *Singleton*, *Dataflow*, or *SyncOrder*, since the loop will neither race nor synchronize with any other part of the program.

Conversely, there are executions that have different data flows or synchronization orders, but terminate with the same state. Examples in $\mathit{FinalState} \cap (\mathit{Dataflow} \setminus \mathit{SyncOrder})$ and $\mathit{FinalState} \cap (\mathit{SyncOrder} \setminus \mathit{Dataflow})$ appear in the proofs of Theorems 3 and 4, respectively, and an example for $\mathit{FinalState} \setminus (\mathit{SyncOrder} \cup \mathit{Dataflow})$ is easy to construct (imagine, for example, a program that has chaotic data flow and synchronization, but eventually writes a zero to every program variable before terminating). □

4.2 Programming Languages and Idioms

While equivalence relations and their relationships, seen from a theoretical perspective, may be interesting in their own right, they probably need to correspond to some intuitively appealing programming language or idiom in order to be of practical interest. As illustrations, we describe five programming idioms in this section, corresponding to the dots labeled A, B, C, D, and E in Figure 1.

Independent Split-Merge. (Point A $\in$ *Singleton* in Figure 1.) Consider a language providing parallel iterators or cobegin, with the requirement (enforced through the type system or run-time checks) that concurrent tasks access disjoint sets of variables. If every task is modeled as a separate thread, then there will be no synchronization or data races, and the execution of a given program on a given input will be uniquely determined.

Bag of Independent Tasks. (Point B $\in \mathit{Dataflow} \setminus \mathit{SyncOrder}$ in Figure 1.) Consider a programming idiom in which "worker" threads dynamically self-schedule independent tasks from a shared bag. The resulting executions will have isomorphic data flow (all that will vary is the tids in the corresponding reads and writes), but their synchronization orders will vary with the order in which they access the bag of tasks.

Significantly, this idiom remains in $\mathit{Dataflow} \setminus \mathit{SyncOrder}$ even if we require that tasks be added to the bag in groups, and all of them completed before any new tasks can be added. One might consider such a restricted model to be an alternative characterization of the Independent Split-Merge idiom, but we prefer to consider it a separate language—one in which the maximum degree of concurrency in the abstract execution is limited to the number of worker threads.

One might also expect that a program with deterministic sequential semantics, no data races, and no synchronization races would have only a single abstract execution for a given input—that is, that $\mathit{Dataflow} \cap \mathit{SyncOrder}$ would equal *Singleton*. We speculate, however, that there may be cases—e.g., uses of rand()—that are easiest to model with more than one execution (i.e., with classically nondeterministic sequential semantics), but that we might still wish to think of as "deterministic parallel programming." We have left a region in Figure 1 (the dark gray area labeled "3") to suggest this possibility.

Parallel Iterator with Reduction. (Point C $\in \mathit{ExternalEvents} \setminus (\mathit{Dataflow} \cup \mathit{SyncOrder})$ in Figure 1.) Consider a language with explicit support for reduction by a commutative, associative function. The order in which such a function is applied to a set of operands need not be fixed, leading to executions with different synchronization orders and data flows, but only a single result. It seems plausible that we might wish to call programs in such a language "deterministic."

Parallel Atomic Commutative Methods. (Point D $\in$ *ExternalEvents* $\setminus$ (*Dataflow* $\cup$ *SyncOrder*) in Figure 1.) In the split-merge and bag-of-tasks idioms above, we required that parallel tasks be independent. We may relax this requirement by allowing tasks to call methods of some shared object O, so long as the calls are atomic and (semantically) commutative. The memory allocator mentioned in Section 1 is an example of this idiom, as long as we ignore the possibility of running out of memory. Another example would be a memoization table that caches outputs of some expensive function.

If a program contains atomic, commutative method calls in otherwise independent tasks, the synchronization order for these calls may be different in different runs of the program on the same input. Data flow may also be different, because the internal state of the shared object may change with the synchronization order. Even the final state may be different, since commutativity is defined at a level of abstraction above that of individual variables. A given finite sequence of calls is guaranteed to lead to the same output, however, regardless of permutation, because the calls are atomic and commutative.

Chaotic Relaxation. (Point E $\in$ *SyncOrder* $\setminus$ *Dataflow* in Figure 1.) Certain spatially partitioned, iterative computations can be proven to converge even if iterations are unsynchronized, allowing local computations to sometimes work with stale data [14]. Execution typically halts once all threads have verified that the error in their local values is less than some threshold ϵ.

Imagine a language that is specially designed for programs of this kind. Programmers can specify an ϵ for the convergence condition, then design an iterative algorithm for an array of data. Different executions may have different data flows, because the program is full of data races. For chaotic relaxation, however, these data races do not change the limit toward which the computation converges. If final results are rounded to a level of significance determined by ϵ before being output, the results will be deterministic despite the uncertainty of data flow. And as long as the output operations (which constitute the only synchronization in the program) are strictly ordered, the program will be deterministic according to *SyncOrder*.

4.3 Repetitive Debugging

One of the principal goals of deterministic parallel programming is to facilitate repetitive debugging. The definitions in Section 3 vary significantly in the extent to which they achieve this goal.

In a *Singleton* system, a debugger that works at the level of abstract executions will be guaranteed, at any breakpoint, to see a state that corresponds to some consistent cut across the happens-before order of the single execution. This guarantee facilitates repetitive debugging, though it may not make it trivial: a breakpoint in one thread may participate in an arbitrary number of cross-thread consistent cuts; global state is not uniquely determined by the state of a single thread. If we allow all other threads to continue running, however, until they wait for a stopped thread or hit a breakpoint of their own, then global state will be deterministic. Moreover (assuming a relatively fine-grain correspondence between target and abstract executions), monitored variables' values will change deterministically, since *Singleton* requires all runs of a program on a given input to correspond to the same abstract execution. This should simplify both debugging and program understanding.

Under *Dataflow*, monitored variables will still change values deterministically, but two executions may not reach the same global state when a breakpoint is triggered, even if threads are allowed to "coast to a stop." A program state encountered in one execution may never arise in an equivalent execution.

Consider the code fragment shown at right (written in a hypothetical language). Assume that f() is known to be a pure function, and that the code fragment is embedded in a program that creates two worker threads for the purpose of executing parallel iterators. In one plausible semantics, the elements of a parallel iteration space are placed in a synchronous queue, from which workers dequeue them atomically.

```
parfor i in [0, 1]
    A[i] = f(i)
print A[0]
print A[1]
```

Even in this trivial example, there are four possible executions, in which dequeue operations in threads 0 and 1, respectively, return $\{0, \perp\}$ and $\{1, \perp\}$, $\{1, \perp\}$ and $\{0, \perp\}$, $\{0, 1, \perp\}$ and $\{\perp\}$, or $\{\perp\}$ and $\{0, 1, \perp\}$. These executions will contain exactly the same operations, except for thread ids. They will have different program and synchronization orders. *Dataflow* will say they are equivalent; *Singleton* will say they are not. If we insist that our programming model be deterministic, *Dataflow* will clearly afford the programmer significantly greater expressive power. On the other hand, a breakpoint inserted at the call to f() in thread 0 may see very different global states in different executions; this could cause significant confusion.

Like *Dataflow*, *SyncOrder* fails to guarantee deterministic global state at breakpoints, but we hypothesize that the variability will be significantly milder in practice: benign data flow changes, which do not impact synchronization or program output, seem much less potentially disruptive than benign synchronization races, which can change the allocation of work among threads.

ExternalEvents and *FinalState*, for their part, offer significant flexibility to the language designer and implementor, but with potentially arbitrary differences in internal behavior across program runs. This would seem to make them problematic for repetitive debugging.

4.4 Deterministic Implementations

Generally speaking, given a deterministic parallel programming language, it should be straightforward to construct an implementation that achieves most of the concurrency that a programmer might expect on a given machine. This expectation is essentially an issue of liveness, and may be difficult to formalize, but the intuition is clear: if the language is capable of expressing only deterministic programs, then an implementation that captures the concurrency explicit in such programs will remain deterministic. In Independent Split-Merge programs, for example, an implementation is assured that synchronization ($<_s$) edges enter a task only at the beginning, and leave it only at the end, so scheduling decisions within a split-merge group can never violate happens-before.

The more interesting question is: in a language that admits nonequivalent abstract executions, how hard is it likely to be (how much run-time cost are we likely to incur) to construct an implementation that achieves a high degree of concurrency (scalability) while still guaranteeing that all target executions will correspond to equivalent abstract executions? Here the answer may depend on just how much nondeterminism the language itself allows. As noted in Section 1, Kendo [26] provides determinism (of roughly the *SyncOrder* variety) only for programs written in the data-race-free subset of C. Specifically, it resolves each synchronization race deterministically, given that deterministic resolution of prior synchronization races and the lack of data races uniquely determines program order in each thread, up to the next synchronization operation. While still too slow for production use (reported overheads are on the order of 1.6×), this is fast enough for convenient repetitive debugging.

For programs with data races, there is no known way to achieve any of our definitions of deterministic implementation without special-purpose hardware or very high worst-case overhead (one can, of course, serialize the execution—we count that as "very high overhead"). CoreDet [5], dOS [6], and Determinator [4] all achieve roughly *Singleton* semantics on conventional hardware, by executing threads in coarse-grain lockstep "epochs," with memory updates applied in deterministic order at epoch boundaries. Unfortunately, all impose common-case overheads of roughly 10×, making them unsuitable for production use and undesirable for debugging. Recent work on the DoublePlay system [31] suggests that it may be possible to execute arbitrary programs deterministically while limiting overhead to a relatively modest amount (comparable to that of Kendo) for executions whose behavior does not depend on data races. (For a brief survey of implementation techniques for deterministic execution, see the recent paper by Bergan et al. [7].)

5 Conclusions and Future Work

Deterministic parallel programming needs a formal definition (or set of definitions). Without this, we will really have no way to tell whether the implementation of a deterministic language is correct. History-based semantics seems like an excellent framework in which to create definitions, for all the reasons mentioned in Section 1. We see a wide range of topics for future research:

– Existing projects need to be placed within the framework. What are their definitions of execution equivalence?
– Additional definitions need to be considered, evaluated, and connected to the languages and programming idioms that might ensure them.
– We need to accommodate condition synchronization, and source-level spinning in particular. Even in *Singleton*, executions that differ only in the number of times a thread checks a condition before finding it to be true should almost certainly be considered to be equivalent.
– We need to decide how to handle operations (e.g., rand()) that compromise the determinism of sequential semantics. Should these in fact be violations? Should they be considered inputs? Should they perhaps be permitted only if they do not alter output?

– Languages embodying the more attractive definitions of determinism should be carefully implemented, and any losses in expressiveness or scalability relative to other definitions carefully assessed.
– We need to examine more thoroughly the issues involved in deterministic implementation of nondeterministic languages.

This final issue seems intriguing. While it may be easy to build an implementation in which all realizable target executions (of a given program and input) correspond to the same abstract execution, such an implementation may be unacceptably slow (e.g., sequential). It may be substantially more difficult to build an implementation that improves performance by exploiting the freedom to realize target executions corresponding to different but nonetheless equivalent abstract executions. This is in essence a question of liveness rather than safety, and it raises a host of new questions: Which executions can be realized by a given implementation? Are certain executions fundamentally more difficult to realize (without also realizing other executions that aren't safe)? What is the appropriate boundary between language- and implementation-level determinism? Progress on these questions, we believe, could significantly enhance the convenience, correctness, and performance of programming in the multicore era.

References

[1] Adve, V., Ceze, L., Ford, B.: Organizers. In: 2nd Workshop on Determinism and Correctness in Parallel Programming, Newport Beach, CA (March 2011)
[2] Adve, S.V., Hill, M.D.: Weak Ordering—A New Definition. In: 17th Intl. Symp. on Computer Architecture, Seattle, WA (May 1990)
[3] Allen, M.D., Sridharan, S., Sohi, G.S.: Serialization Sets: A Dynamic Dependence-Based Parallel Execution Model. In: 14th ACM Symp. on Principles and Practice of Parallel Programming, Raleigh, NC (February 2009)
[4] Aviram, A., Weng, S.-C., Hu, S., Ford, B.: Efficient System-Enforced Deterministic Parallelism. In: 9th Symp. on Operating Systems Design and Implementation, Vancouver, BC, Canada (October 2010)
[5] Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D.: CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In: 15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, PA (March 2010)
[6] Bergan, T., Hunt, N., Ceze, L., Gribble, S.D.: Deterministic Process Groups in dOS. In: 9th Symp. on Operating Systems Design and Implementation, Vancouver, BC, Canada (October 2010)
[7] Bergan, T., Devietti, J., Hunt, N., Ceze, L.: The Deterministic Execution Hammer: How Well Does It Actually Pound Nails? In: 2nd Workshop on Determinism and Correctness in Parallel Programming, Newport Beach, CA (March 2011)
[8] Berger, E., Yang, T., Liu, T., Novark, G.: Grace: Safe Multithreaded Programming for C/C++. In: OOPSLA 2009 Conf. Proc., Orlando, FL (October 2009)
[9] Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing Transactional Semantics: The Subtleties of Atomicity. In: 4th Workshop on Duplicating, Deconstructing, and Debunking, Madison, WI (June 2005)
[10] Bocchino Jr., R.L., Adve, V.S., Dig, D., Adve, S., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A Type and Effect System for Deterministic Parallel Java. In: OOPSLA 2009 Conf. Proc., Orlando, FL (October 2009)

[11] Boehm, H.-J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: SIGPLAN 2008 Conf. on Programming Language Design and Implementation, Tucson, AZ (June 2008)
[12] Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., Taşirlar, S.: Concurrent Collections. Journal of Scientific Programming 18(3–4) (August 2010)
[13] Ceze, L., Adve, V.: Organizers. In: Workshop on Deterministic Multiprocessing and Parallel Programming, Seattle, WA (November-December 2009)
[14] Chazan, C., Miranker, W.: Chaotic Relaxation. Linear Algebra and Its Applications 2, 199–222 (1969)
[15] Dalessandro, L., Scott, M.L., Spear, M.F.: Transactions as the Foundation of a Memory Consistency Model. In: 24th Intl. Symp. on Distributed Computing, Cambridge, MA (September 2010); Previously Computer Science TR 959, Univ. of Rochester (July 2010)
[16] Devietti, J., Lucia, B., Ceze, L., Oskin, M.: DMP: Deterministic Shared Memory Multiprocessing. In: 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Washington, DC (March 2009)
[17] Emrath, P.A., Padua, D.A.: Automatic Detection of Nondeterminacy in Parallel Programs. In: ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI (May 1988)
[18] Halstead Jr, R.H.: Multilisp: A Language for Concurrent Symbolic Computation. ACM Trans. on Programming Languages and Systems 7(4), 501–538 (1985)
[19] Herlihy, M.P., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. on Programming Languages and Systems 12(3), 463–492 (1990)
[20] Hower, D.R., Hill, M.D.: Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In: 35th Intl. Symp. on Computer Architecture, Beijing, China (June 2008)
[21] Karp, R.M., Miller, R.E.: Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. SIAM Journal on Applied Mathematics 14(6), 1390–1411 (1966)
[22] Manson, J., Pugh, W., Adve, S.: The Java Memory Model. In: 32nd ACM Symp. on Principles of Programming Languages, Long Beach, CA (January 2005)
[23] Midkiff, S., Pai, V., Bennett, D.: Organizers. In: Workshop on Integrating Parallelism Throughout the Undergraduate Computing Curriculum, San Antonio, TX (February 2011)
[24] Montesinos, P., Ceze, L., Torrellas, J.: DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In: 35th Intl. Symp. on Computer Architecture, Beijing, China (June 2008)
[25] Netzer, R.H.B., Miller, B.P.: What Are Race Conditions? Some Issues and Formalizations. ACM Letters on Programming Languages and Systems 1(1), 74–88 (1992)
[26] Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: Efficient Deterministic Multithreading in Software. In: 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Washington, DC (March 2009)
[27] Papadimitriou, C.H.: The Serializability of Concurrent Database Updates. Journal of the ACM 26(4), 631–653 (1979)
[28] Shavit, N.: Organizer. In: Workshop on Directions in Multicore Programming Education, Washington, DC (March 2009)
[29] Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: 12th Intl. Conf. on Principles of Distributed Systems, Luxor, Egypt (December 2008)
[30] Steele Jr., G.L., Saraswat, V.A.: Organizers. In: Workshop on Curricula for Concurrency, Orlando, FL (October 2009)
[31] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P., Flinn, J., Narayanasamy, S.: DoublePlay: Parallelizing Sequential Logging and Replay. In: 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Newport Beach, CA (March 2011)

CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention

Dmitry Basin[1], Rui Fan[2], Idit Keidar[1], Ofer Kiselov[1], and Dmitri Perelman[1,⋆]

[1] Department of Electrical Engineering, Technion, Haifa, Israel
{sdimbsn@tx,idish@ee,sopeng@t2,dima39@tx}.technion.ac.il
[2] School of Computer Engineering, Nanyang Technological University
fanrui@ntu.edu.sg

Abstract. Task pools have many important applications in distributed and parallel computing. Pools are typically implemented using concurrent queues, which limits their scalability. We introduce *CAFÉ, Contention and Fairness Explorer*, a scalable and wait-free task pool which allows users to control the trade-off between fairness and contention. The main idea behind CAFÉ is to maintain a list of *TreeContainers*, a novel tree-based data structure providing efficient task inserts and retrievals. TreeContainers don't guarantee FIFO ordering on task retrievals. But by varying the size of the trees, CAFÉ can provide any type of pool, from ones using large trees with low contention but less fairness, to ones using small trees with higher contention but also greater fairness.

We demonstrate the scalability of TreeContainer by proving an $O(\log^2 N)$ bound on the step complexity of insert operations when there are N inserts, as compared to an average of $\Omega(N)$ steps in a queue based implementation. We further prove that get operations are wait-free. Evaluations of CAFÉ show that it outperforms the Java SDK implementation of the Michael-Scott queue by a factor of 30, and is over three times faster than other state-of-the-art non-FIFO task pools.

1 Introduction

A *task pool* is a data structure consisting of an unordered collection of objects, a *put* operation to add an object to the collection, and a *get* operation to remove an object[1]. Pools have a number of important applications in multiprocessor computing, such as maintaining the set of pending tasks in a parallel computation. A key challenge in such an application is to ensure the pool does not become a bottleneck when it is concurrently accessed by a large number of threads. Another challenge is to ensure fairness — although strict FIFO ordering is not necessary, we nevertheless want to avoid starvation and limit the number of *overtakings*[2].

In this paper, we present CAFÉ (Contention And Fairness Explorer), an efficient randomized wait-free[3] task pool algorithm. CAFÉ maintains a list of scalable bounded

⋆ This work was partially supported by Hasso Plattner Institute.

[1] We sometimes refer to task pools as producer-consumer pools; producers do puts, and consumers do gets.

[2] One task overtakes another task if it is inserted after the other task, but retrieved before it.

[3] A randomized algorithm is *wait-free* if each thread executing an operation performs a finite number of steps with probability 1.

pools called *TreeContainers*. When one TreeContainer becomes full, a new TreeContainer is appended to the end of the list. Retrievals follow the FIFO order of the TreeContainers, but each TreeContainer can return its tasks in any order. This way, the tree size is a system parameter controlling the trade-off between fairness and contention. Using smaller trees, the system provides better fairness but also has more contention.

A TreeContainer stores jobs in a complete binary tree, in which every node can store one task. Each node keeps presence bits indicating whether its child subtrees contain tasks. This allows get operations to find tasks by walking down the tree from the root, following a trail of presence bits. At the same time, the bits do not change frequently, even when there are a large number of concurrent puts and gets, so they do not cause much contention. We show that TreeContainers are dense: a tree with height h contains at least $2^{(1-\epsilon)h}$ tasks with high probability, for any $\epsilon > 0$. We also show that TreeContainers perform well under contention. When there are N concurrent put operations and an arbitrary number of gets, each put finishes in $O(\log^2 N)$ steps, whp.

CAFÉ combines TreeContainers in a FIFO linked list, to provide the following properties. 1) The number of overtaken tasks in CAFÉ is bounded by the size of a tree. 2) In most workloads, producers and consumers operate on different TreeContainers, which decreases contention and improves performance. 3) Puts are wait-free with probability 1, and gets are deterministically wait-free.

Our algorithm offers some significant advantages over other approaches for task pools. The most common approach to implement pools is using FIFO queues (e.g., Java ThreadPoolExecutor). However, non-blocking queue-based algorithms suffer $\Omega(N)$ contention at the head and tail, while our algorithm has $O(\log^2 N)$ contention for puts, whp. Other queue-based algorithms are blocking, and require puts and gets to wait for each other. In contrast, all operations in our algorithm are wait-free. The recent ED pools in [1] also use trees, but in a different way. Unlike our algorithm, [1] does not provide any upper bounds on step complexity, nor on the number of times a task can be overtaken.

We have implemented CAFÉ in Java, and tested its performance on a 32-core machine[4]. Our results show that CAFÉ is over 30 times faster than a pool based on Java's implementation of the Michael-Scott queue, and over three times faster than a pool using Java's state-of-the-art blocking queue (even though CAFÉ does not block). Also, CAFÉ is over three times faster than ED pools, while providing stronger fairness guarantees.

The remainder of the paper is organized as follows. In Section 2, we describe related work. We present CAFÉ in Section 3, and analyze its theoretical properties in Section 4. We discuss our experimental results in Section 5. Finally, we conclude in Section 6.

2 Related Work

A common approach to implementing concurrent task pools is to use FIFO queues for task management. However, due to their strong ordering guarantees, such implementations are not scalable, suffering from $\Omega(N)$ contention in the worst case. CAFÉ makes

[4] The code is publicly available at `http://code.google.com/p/cafe-pool/`

the observation that strict FIFO ordering is not necessary for a task pool, and thereby achieves a much more scalable algorithm.

Another approach for reducing contention is using *elimination*, as proposed by Moir et al. [8]. Here, producers and consumers can "eliminate" each other at predefined rendezvous points. This approach best suits workloads in which there are more consumers than producers. Elimination is less useful if the queue remains non-empty most of the time, or when concurrency is low. In contrast, CAFÉ performs well under both high and low concurrency, and regardless of the ratio between producers and consumers[5].

Afek *et al.* [1] also propose a task pool foregoing FIFO ordering for scalability. Their Elimination Diffraction (ED) pools yield significantly better results than FIFO implementations. ED pools use a fixed number of queues along with elimination for reducing contention. However, as we show in Section 5.2, ED pools do not scale well on multi-chip architectures. In addition, unlike CAFÉ , ED pools are not wait-free, and offer no fairness guarantees between puts and gets.

The idea of using concurrent tree-based data structures for reducing contention has appeared in previous works not related to task pools [4,3]. Unlike these works, we prove formal bounds on the worst case step complexity of our TreeContainer algorithm.

3 CAFÉ: A Task Pool with Adjustable Fairness

In this section, we describe CAFÉ, a wait-free, scalable task pool algorithm, whose fairness can be adjusted arbitrarily by the user. The main idea behind CAFÉ is to keep a linked list of scalable task pools called *TreeContainers*, each with bounded size. The algorithm for a single TreeContainer is given in Section 3.1. Tasks are stored at tree nodes, which can be occupied at most once. When a tree becomes full, a new tree is added to the list. The algorithm for combining TreeContainers in a FIFO list is described in Section 3.2.

3.1 TreeContainer

A TreeContainer consists of a bounded complete binary tree, in which each node can store one task. A node with a task is *occupied*, and otherwise it is *free*. Each node can be occupied at most once, as indicated by an *isDirty* flag. In addition, the node keeps a *presence bit* for each child subtree; the bit is zero when all the nodes in the respective subtree are free. Presence bits allow get operations to find a task in the tree by walking down from the root following a trail of non-zero bits. Since presence bits summarize the occupancy of an entire subtree, they change infrequently even under highly concurrent workloads, which allows our algorithm to achieve low step complexity.

TreeContainer is shown in Algorithm 1. Level i of the tree is implemented using an array tree[i], which allows $O(1)$ access to any node in a level. The root is the only node at level 0. Each node also keeps pointers to its father and children, as well as a bit *side*, indicating whether it is the left or right child of its father.

[5] Due to space limitations, evaluations of CAFÉ on different workloads is deferred to the full paper [2].

Algorithm 1. TreesContainer, a scalable bounded task pool algorithm

```
 1: TreeNode data structure:
        ▷ ver: version of the metadata
        ▷ p indicates presence of tasks in left/right subtrees
        ▷ ⟨ver, p⟩ is kept by a single AtomicInteger in Java
 2:     [⟨ver, p⟩, ⟨ver, p⟩]: meta
 3:     int: pending
 4:     boolean: isDirty        ▷ true if has been already used
 5:     Data: task
 6:     int: side     ▷ 0 for the left child, 1 for the right child
 7: Tree data structure:
        ▷ tree[$i$] keeps an array with the nodes of level $i$
 8:     TreeNode[][]: tree

 9: Function hasTasks(node):
10:     if (node.meta[0].p $\vee$ node.meta[1].p)
11:         then return 1
12:     else return (node.task $\neq \perp$) ? 1 : 0

13: Function put(task):
14:     node $\leftarrow$ findNodeForPut(task)
15:     if (node $= \perp$) then return false
16:     updateNodeMetadata(node, 1)
17:     return true

18: Function findNodeForPut(task):
19:     for level $= 0, 1, \ldots$ do
20:         trials $\leftarrow$ (level $<$ height(root)) ? 1 : $k$
21:         for i $= 1, \ldots,$ trials do
22:             node $\leftarrow$ random node in tree[level]
23:             reserved $\leftarrow$ putInNode(node,task)
24:             if (reserved $\neq \perp$) return reserved
25:     return $\perp$                  ▷ did not succeed in this tree

26: Function putInNode(node, task)
27:     if (node.father $\neq \perp \wedge$ node.father.task $= \perp$)
28:         return putInNode(node.father, task)
29:     if (node.isDirty.CAS(false, true))
30:         node.task $\leftarrow$ task; return node
31:     else return $\perp$

32: Function get()
33:     while(true):
34:         if (hasTasks(root) $= 0$) return $\perp$
35:         node $\leftarrow$ findNodeForGet()
36:         task $\leftarrow$ node.task
37:         if (task $\neq \perp \wedge$
                node.task.CAS(task, $\perp$) $=$ false) continue
38:         updateNodeMetadata(node, 0)
39:         if (task $\neq \perp$) return task

40: Function findNodeForGet()
41:     node $\leftarrow$ root
42:     while(true)
43:         if(node.task$\neq\perp \vee$
                node.meta[0].p=node.meta[1].p=0)
44:             return node
45:         node $\leftarrow$ random child among those with p $= 1$

46: Function updateNodeMetadata(node, myVal)
47:     trials $\leftarrow$ 0;
48:     while(node.father $\neq \perp$)
            ▷ check if my operation has been eliminated
49:         if (myVal $\neq$ hasTasks(node)) return
50:         fk $\leftarrow$ father.meta[node.side].p
51:         if (fk $\neq$ hasTasks(node) $\vee$ node.pending $>$ 0)
52:             trials $\leftarrow$ trials +1
53:             if (updateFather(node) $\neq$ success $\wedge$
                    trials $<$ 2) continue          ▷ try again
54:         node $\leftarrow$ node.father; trials $\leftarrow$ 0

55: Function updateFather(node)
56:     node.pending.FetchAndInc()
57:     new $\leftarrow$ old $\leftarrow$ father.meta[node.side]
58:     new.ver $\leftarrow$ new.ver +1; new.p $\leftarrow$ hasTasks(node)
59:     success $\leftarrow$ father.meta[node.side].CAS(old, new)
60:     node.pending.FetchAndDec()
61:     return success
```

Task Insertion. Tasks are inserted in a tree using the $put()$ operation. First, put finds a free node to insert the task. Then it updates the presence bits of the node's ancestors. Because a tree has bounded size, task insertions can fail if they do not find a free node in the tree. Below, we describe the main steps in a put.

Finding an unoccupied node. Function *findNodeForPut()* finds a free tree node for task insertion. It iterates over the tree levels starting from the root (lines 19–24). At each level, a random node x is chosen, and the algorithm tries to put the task in the *highest* free node on the path from x to the root. This is done using the recursive function *putInNode()* (lines 27–31). Nodes are reserved by CASing the $isDirty$ flag. Having nodes search for a free ancestor increases put's step complexity from $O(h)$ to $O(h^2)$ for a tree with height h (proved in [2]). However, it also creates denser trees with a more balanced node occupation, as we show in Section 4.2 and prove in [2].

If neither x nor its ancestors can be reserved, another random node is checked. At each level except the last one, a single node is checked. The number of nodes checked at the last level is defined by a parameter k, with higher k's resulting in denser trees. We show in Section 4.2 and prove in [2] that in a tree with height h, at least $2^{\frac{k+2}{k+3}h}$ nodes are occupied before a put operation fails, whp.

Updating ancestors' metadata. After a task is inserted in node x, function *updateNodeMetadata()* updates the presence bits of x's ancestors (lines 48–54). At each node the function checks that the metadata of the father is correct. Contention remains low because in the common case, the presence bits of upper level nodes are not updated when a new task is inserted or removed.

Though the general outline of the algorithm is simple, ensuring linearizability, wait-freedom and low contention require special care, as we describe below.

1. *Ensuring linearizability.* A naïve approach to update x's father's metadata could be to first read the old presence bit of x's father (line 50), then calculate whether x's subtree contains tasks (line 57), and finally CAS a new metadata value if the old value is incorrect (line 59). If the CAS fails, the updater retries. Version numbers are attached to the presence bits in order to avoid ABA problems.

Unfortunately, this simple approach can violate linearizability. Consider nodes x, y and z, where y is the right child of x and z is the right child of y. Node y has a task, so that $x.meta[1].p = 1$. There are two concurrent threads, a consumer t_c that removes the task from y and a producer t_p that inserts a task in z. t_c starts updating the metadata of y's father. It reads the right presence bit at x, which is 1, and decides to update it to 0. We then suspend t_c right before it performs its CAS operation. At this time, t_p starts updating the ancestors of z. It first changes $y.meta[1].p$ from 0 to 1, and then checks the right presence bit at x. Since t_c is paused, $x.meta[1].p$ is still 1, and so t_p decides this value is correct, and terminates. Now t_c resumes, and successfully changes $x.meta[1].p$ to 0. This makes future gets think the tree is empty, so that no get will retrieve t_p's task, violating linearizability.

We solve this problem by letting other threads know about concurrent pending updaters. Whenever a thread t plans to change the metadata of x's father, it increments a *pending* counter at x (line 56); after the update, it decrements the counter (line 60). If a concurrent updater sees $x.pending > 0$, it will update x's father's metadata, regardless of its current value (line 51). This, along with the use of version numbers, will cause the pending thread's CAS to later fail.

2. *Limiting the number of CAS failures.* In the simple algorithm described earlier, an updater thread t that fails to CAS the metadata of x's father will retry the update. This makes t's worst case step complexity linear in the tree size, since every thread that successfully performed an operation in x's subtree can cause t's CAS to fail. However, as we show in the full version of the paper [2], it suffices for t to only try to update x's father's metadata *twice* (line 53). The idea is that if t fails two CASes, then some other thread will have already updated x's father's metadata to the correct value.

3. *Producer/consumer elimination.* We have also adopted the elimination technique used in [8] and [1]. Consider a thread t that inserted a new task at a node, and started updating the node's ancestors. Let x and y be two such ancestors, where y is the father of x. In the function $updateNodeMetadata$, t updated y's metadata (on x's side) to 1 while t was still at x. Thus, if t later arrives at y and sees y's x-side metadata is now 0, it means there has been consumer thread that already removed the task t inserted. In this case, t doesn't need to update any more ancestors, and can terminate early (line 49). This optimization improves performance in scenarios where multiple producers and consumers are working on the same tree.

We show in Section 4.2 and prove in [2] that put operations in TreeContainer are wait-free. Intuitively, this is because the tree is bounded, and because a thread only tries two updates per node. If the tree has height h, the put performs $O(h^2)$ steps. We show in Section 4.2 that our insertions create a balanced tree, whp. Hence, when the tree contains N tasks, the complexity of a put is $O(\log^2 N)$.

Task Retrieval. The *get()* function in TreeContainer runs in a loop (lines 33–39). If there are no tasks in the tree, as indicated by the presence bits at the root, the function returns $\perp$ (line 34). $get()$ first finds a task at a random node to retrieve from using $findNodeForGet()$, and then updates the metadata of the node's ancestors.

Function *findNodeForGet()* searches for a node to get a task from. When it reaches an unoccupied node, it randomly chooses a nonempty subtree to go down. The randomization reduces contention.

A task T is removed from node x by CASing $x.task$ from T to $\perp$ (line 37). If the CAS succeeds, then the metadata of x's ancestors need to be updated. Otherwise, the algorithm starts a new retrieval attempt. Note that if *findNodeForGet()* finds a node x with $x.task = \perp$, it means that another consumer t_c removed x's task but still hasn't updated x's ancestors. In order to be wait-free, a consumer needs to make sure that it will not arrive to this empty node infinitely many times. Hence, a consumer that arrives at an empty node x updates x's ancestors even though it did not take x's task (line 38). Updating the ancestors is done the same way as after a task insertion, using *updateNodeMetadata()*.

We show in Section 4.2 and prove in [2] that get operations are wait-free. Intuitively, this is because a get thread t_c can only fail to take a task from a previously occupied node x if some other thread took x's task. Then, t_c updates the metadata on the path to the root, so that t_c does not go down the same path again. The bounded number of nodes in a tree then limits the number of unsuccessful get attempts.

3.2 Combining TreeContainers in a FIFO List

As stated earlier, CAFÉ maintains a linked list of TreeContainers, adding new trees as old ones become full (see Figure 1). Tasks are returned in FIFO order, up to the tree they are inserted into. This guarantees that the maximum number of overtakers in CAFÉ is bounded by the tree size. Therefore, the tree size is a parameter that determines the trade-off between fairness and contention. Using bigger trees, CAFÉ performs more like a TreeContainer, and so has low contention but less fairness. Using smaller trees, CAFÉ performs more like a FIFO list, so there is higher contention but greater fairness.

Basic approach. A simple way to manage a linked list of trees is to keep one pointer (PT) for producers, which references the tree for puts, and another (GT) for consumers, referencing the tree for gets. Whenever the current insertion tree becomes full, PT is moved forward. Whenever no tasks are left in the retrieval tree, GT is moved forward. Old trees are garbage collected automatically in managed memory systems as they become unreachable.

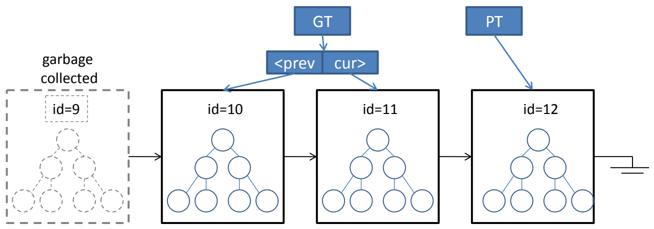


Fig. 1. CAFÉ keeps a linked list of scalable task trees. The tree height defines the fairness of the protocol.

This straightforward approach, however, violates correctness, as we now demonstrate. Consider the following scenario. t_p inserts a task in tree T and pauses before changing the metadata of T's root. Consumers assume that T is empty and increment GT to continue to later trees. When t_p finally resumes, we have $GT > PT$, and no consumer will ever retrieve t_p's task.

One way to solve this problem is to reinsert the task in a later tree whenever t_p notices its task may be lost. However, this approach might lead to livelocks, in which producers constantly chase consumers, never finishing their operations. Another method is to maintain a non-zero indicator on each tree (e.g., using SNZI [3]) indicating whether there are concurrent producers working on the tree. But this approach incurs high overhead, for managing both indicators and lists of "pending and active" trees. Our solution is instead based on the idea of moving the consumer pointer GT backwards when a task is added in an old tree.

Managing the list of trees. The pseudo-code for the list of trees pool is shown in Algorithm 2. A put operation tries to insert the task into the tree pointed to by PT (call this tree T). If the insert fails, the algorithm moves to the next tree in the list by incrementing PT (lines 16–17). New trees are created and appended to the end of the list as needed. For reasons we explain later, the pointer for consumers GT actually points to two consecutive trees, $GT.cur$ and $GT.prev$. When an insert succeeds, the producer checks that its task will be retrievable in the future. To this end, it checks that $GT.cur$ does not point to a tree that succeeds T in the linked list (line 13). If it does, the GT pair is moved backwards to $\langle \perp, T \rangle$ in the function $moveGTBack$.

In $moveGTBack$, a producer repeatedly tries to CAS GT to T until a CAS succeeds, or it reads $GT.cur \leq T$. As we want producers to be wait-free, we need to ensure this loop eventually terminates. Thus, we do not allow the GT pointers to move forward while there are pending producers that want to move GT backwards. We increment a counter $oldProducers$ at the start of $moveGTBack$, and decrement it at the end. If a consumer does not find a task in the GT trees, but sees $oldProducers > 0$, it advances to a later tree, but does not increment GT (line 44).

A consumer tries to retrieve a task from the trees pointed to by $GT.prev$ and $GT.cur$ (lines 36–37). If both trees are empty, and if PT points to a later tree than $GT.cur$, then GT is updated to $\langle GT.cur, GT.cur.next \rangle$. This update is performed by first creating a pair with the new tuple values (line 40), and then CASing GT from the old pair to the

Algorithm 2. CAFÉ algorithm for adjustable fairness and contention

```
 1: Data structures:
 2:     Node:
 3:         int: id
 4:         ScalableTree: tree
            Node: next
 5: Global variables:
 6:     Node: PT                          ▷ tree for producers
 7:     ⟨prev, cur⟩: GT                   ▷ tree for consumers
 8:     int: oldProducers                 ▷ for moving GT backwards
 9: Function put(task)
10:     while(true)
11:         latest ← PT
12:         if (latest.tree.put(task) = true) then
13:             if (GT.cur.id > latest.id)
                        moveGTBack(latest)
14:             return
15:         else
16:             if(latest.next = ⊥) insertNewTree()
17:             PT.CAS(latest, latest.next)
18: insertNewTree()
19:     newNode ← Node()
20:     cur ← PT ▷ go to the end of the list
21:     for(; cur.next ≠ ⊥; cur ← cur.next);
22:     newNode.id ← cur.id +1
23:     cur.next.CAS(⊥, newNode)  ▷ return even if CAS
                                    fails
24: Function moveGTBack(Node: prodTree)
25:     oldProducers.FetchAndInc()
26:     while(true)
27:         gtVal ← GT
28:         if (gtVal.cur.id ≤ prodTree.id) break
29:         newGT ← ⟨⊥, prodTree⟩
30:         if (GT.CAS(gtVal, newGT) = true) break
31:     oldProducers.FetchAndDec()
32: Function get()
33:     ptVal ← PT
34:     gtVal ← GT
35:     while(true)
36:         task ← gtVal.prev.getTask()
            if (task ≠ ⊥) return task
37:         task ← gtVal.cur.getTask()
            if (task ≠ ⊥) return task
            ▷ could not find a task in the tree
38:         if (ptVal.id ≤ gtVal.cur.id) return ⊥
39:         if (oldProducers = 0) then
40:             newGT ← ⟨gtVal.cur, gtVal.cur.next⟩
41:             GT.CAS(gtVal, newGT)
42:             gtVal ← GT
43:         else
44:             gtVal ← ⟨gtVal.cur, gtVal.cur.next⟩
```

new one (line 41). Note that the ABA problem does not occur during the CAS, because every newly created pair is a new object whose address is different from the addresses of any old pairs, which are not deallocated throughout the function's execution.

Finally, we explain the reason for using two consumer pointers, $GT.cur$ and $GT.prev$. Suppose GT only pointed to one tree, and consider the following situation. GT and PT both point to a tree T. Producer t_p inserts a new task in T and pauses. Meanwhile, other producers insert new tasks, append new trees and move PT. Suppose a consumer t_c comes to retrieve a task, does not find any tasks in T, and pauses right before changing GT to $T.next$. When t_p resumes, it inserts its task to T, checks that GT is still pointing to T and terminates. When t_c resumes, it changes GT to $T.next$. Now, t_p's task is lost. As we show in the next section, keeping two pointers allows us to solve this problem in a simple and efficient way.

In the next section, we show that both put and get operations in CAFÉ terminate within a finite number of steps with probability 1. Thus, CAFÉ is wait-free.

4 CAFÉ's Properties

In this section, we present the correctness and performance properties of CAFÉ. Due to space limitations, we only state the main results and describe the ideas behind them, deferring the full proofs to the full paper [2]. For all the results we assume that an adversary controls thread scheduling but cannot influence the randomness threads use. We let h denote the height of a TreeContainer, and k denote the number of insertion attempts in the last layer of TreeContainer (line 20 in Algorithm 1).

4.1 Safety Properties

In this section we present safety proof outline. We start by showing that CAFÉ implements a linearizable job pool. Intuitively, if the job pool is nonempty, then a get must be able to find a job. We prove a theorem showing that after any put operation finishes, no subsequent get operation will return $\perp$, until the put's task has been returned.

Theorem 1. *Suppose a get operation g in CAFÉ returns $\perp$ at a time t. Then for every put operation p that completed before the start of g, p's task was removed by some get operation before t.*

The Theorem 1 proof consists of two parts. First, we prove that each TreeContainer CAFÉ uses is itself a linearizable job pool. We formalize this in Lemma 1.

Lemma 1. *TreeContainer implements a linearizable producer-consumer pool.*

Second, we prove that after a put inserts a task in some TreeContainer, subsequent get operations will not skip this TreeContainer when looking for a job. We formalize this in Lemma 2.

Lemma 2. *Let p be a completed put operation that inserted a task in TreeContainer T. Suppose at some time τ, p's task has not been removed. Then $GT.cur.id \leq T.id + 1$ at τ.*

The key to proving Lemma 1 is Lemma 3.

Lemma 3. *Consider any TreeContainer T, and let p be a completed put operation that inserted a task in node $x_0 \in T$. Suppose that by some time τ, no get operation has removed the task from x_0,* i.e. *line 37 with $node = x_0$ has not occurred (Algorithm 1). Then for every node x on the path from x_0 to the root of T, $hasTasks(x) = 1$ at τ.*

The lemma proves that after a put operation has inserted a task in some node of a TreeContainer, $hasTasks(x) = 1$ for every node x on the path from that node to the root of the TreeContainer, until the node's task is removed. We say that the nodes on the path are *marked*. Get operations follow a path of marked nodes, and so will always find a job as long they have not all been removed. We briefly describe the proof of Lemma 3. Let x and y be two nodes a put operation p passes through during $updateNodeMetadata$, where y is the father of x. The invariant we maintain is that the value of $hasTasks(x)$ has been fixed to 1 by the time p starts updating y's metadata. Since p tries to set y's metadata to $hasTasks(x)$, then $hasTasks(y)$ will also be fixed to 1 after p finishes processing y. Thus, all the $hasTasks$ values on the path from p's insertion node to the root will be fixed to 1 inductively.

Next, we briefly describe the proof of Lemma 2. After a put operation has inserted a task in a tree T, it does *moveGTBack* to ensure the value of GT is at most T. There are two ways the put checks this condition. Either it successfully CASed the value $\langle \perp, T \rangle$ into GT, or it read that $GT.cur$ is at most T. Because the CASes on GT can be linearized, we can show in the first case that later gets see T (or a smaller value) when they read GT. In the second case, we need to be careful that while the put is checking $GT.cur$ is at most T, there may be a paused get operation, which then increases GT as soon as the put's check finishes. However, even if this happens, $GT.cur$ only moves forward by 1. Since a get operation checks both $GT.cur$ and its preceding tree $GT.prev$, the get will still see the tree that the put inserted into.

The last correctness property we show is that gets return jobs in FIFO order, up to the TreeContainer they were inserted into. This follows simply because jobs are inserted and removed based on the linked list order of the TreeContainers.

4.2 Performance Properties

We first show that our trees are dense: by choosing an appropriate k we can guarantee that a tree with height h is populated with at least $2^{(1-\epsilon)h}$ tasks for an arbitrary $0 < \epsilon < 1$, with high probability. In the full paper [2], we also show that this density is higher than that achieved by a simple random walk based insertion. More formally, we prove the following lemma.

Lemma 4. *In a TreeContainer of height h, if a put operation fails, then the tree contains at least* $2^{\frac{k+2}{k+3} \cdot h}$ *tasks with probability at least* $1 - \frac{1}{2^{(3-\frac{7}{k+3})h+k+1}}$.

In addition, we prove that TreeContainer has a bound on put operation step complexity:

Lemma 5. *Every* put() *operation of TreeContainer makes at most* $O(h^2)$ *steps.*

We further demonstrate that TreeContainer performs well under contention. For N concurrent put operations and an arbitrary number of get operations, each put finishes in $O(\log^2 N)$ steps, whp:

Lemma 6. *Consider a TreeContainer after* N *successful* put *operations. Then each of these operations has taken* $O(\log^2 N)$ *steps with probability at least* $1 - \frac{1}{2(N+1)^{\frac{4}{3}}}$.

We next intuitively demonstrate the wait-freedom of CAFÉ. We first show that put operations are wait-free with probability 1, and then argue that get operations are deterministically wait-free.

A put operation traverses the linked list of TreeContainers until it successfully inserts a task in one of them; new TreeContainers are appended if the insertions keep failing. Intuitively, it might seem that this traversal could go on forever. For example, a slow thread t_p could repeatedly try to insert a task in some tree, then pause until all other producers proceed to a new tree, fail its current insert, and have to retry in a new tree. Fortunately, this situation does not happen. Due to the randomness in the algorithm, other threads are likely to have left unoccupied nodes in t_p's tree, which t_p can acquire once it resumes. We formalize this intuition in the following lemma.

Lemma 7. *If P producer threads and any number of consumer threads use CAFÉ, then any TreeContainer's* put *operation succeeds with probability at least* $(1 - \frac{1}{2^h})^{k(P-1)} \cdot [1 - (1 - \frac{1}{2^h})^k]$.

Using Lemma 7, we prove the following. Note that CAFÉ using TreeContainers of height 0 is equivalent to a linked list.

Lemma 8. *If the height of TreeContainer is greater than zero, then CAFÉ's* put *operations are wait-free with probability* 1.

In order to show CAFÉ 's get operations are wait-free, we need to show that a consumer does not need to traverse an unbounded number of trees when looking for a task. This is true because each get operation keeps a pointer to the latest TreeContainer when it starts (line 33 in Algorithm 2), and subsequently only checks trees that had tasks before it started. In a linearizable execution, the get is allowed to return $\perp$ when all these trees are empty (in line 38), as all their tasks will have been taken by other gets concurrent with or preceding the current get. We conclude with the following lemma.

Lemma 9. *Every get operation of CAFÉ terminates in a finite number of steps.*

5 Evaluation

In this section we evaluate the performance of Java implementation of CAFÉ. Due to space limitations, we only describe the highlights of our evaluation. More comprehensive experimental results may be found in the full paper [2].

5.1 Experiment Setup

We compare the following task pool implementations:

- **CAFÉ-h** – CAFÉ with height h for each tree. Unless stated otherwise, we use $h = 12$.
- **CLQ** – The standard Java 6 implementation of a (FIFO) non-blocking queue by Michael and Scott [7] (class `java.util.concurrent.ConcurrentLinkedQueue`, which is considered to be one of the most efficient non-blocking algorithms in the literature [5,6].
- **LBQ** – The standard Java 6 implementation of a (FIFO) blocking queue that uses a global reader-writer lock (class `java.util.concurrent.LinkedBlockingQueue`).
- **ED** – The original elimination-diffraction tree implementation [1] (downloaded from the web page of the project), in its default configuration. Tasks are inserted into a diffraction tree with FIFO queues attached to each leaf. The queues are implemented using Java LinkedBlockingQueues. Every tree node contains an elimination array where producers can pass tasks directly to consumers. Changing the tree depth, pool size and spinning behavior did not have a significant effect on the pool's performance. Note that ED trees, like CAFÉ , do not enforce FIFO ordering.

We use a synthetic benchmark for the performance evaluation, in which producer threads work in loops inserting dummy items, and consumer threads work in loops retrieving dummy items.

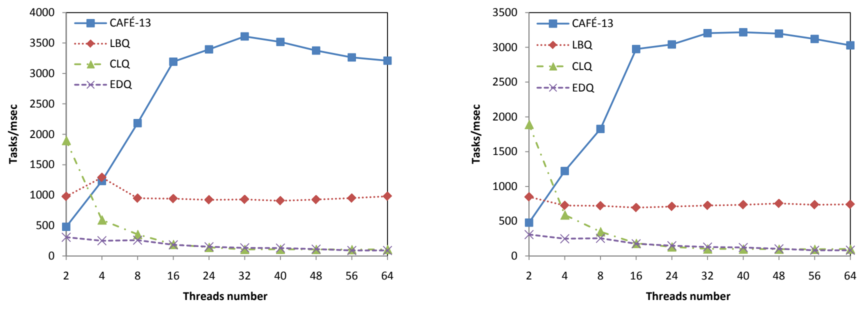
Unless stated otherwise, tests are run on a dedicated shared memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor. JVM is run with the AggressiveHeap flag on. We run up to 64 threads on the 32 cores. The influence of garbage collection was negligible for all algorithm[6].

[6] This was checked using the verbose:gc flag in JVM.

We analyze system performance in Section 5.2 and study the influence of tree heights in Section 5.3.

5.2 System Throughput

In Figure 2 we show the average insertion and retrieval rates in a system with an equal number of producers and consumers. Both graphs demonstrate the same behavior. The throughput of CAFÉ increases up to 32 threads, the number of hardware threads in our architecture. At this point, the throughput of CAFÉ is $\times 30$ higher than the Michael-Scott queue or the ED pool. It is also over three times higher than the blocking queue. When the number of working threads exceeds the number of hardware threads in the system, the throughput of CAFÉ decreases moderately, but still outperforms the other algorithms.



(a) Task insertion rate (b) Task removal rate

Fig. 2. Task insertion and retrieval rates (equal numbers of producers and consumers). The throughput of CAFÉ-13 increases up to 32 threads (the number of hardware threads in the system). In this configuration it is $\times 30$ faster than the Michael-Scott ConcurrentLinkedQueue and over three times higher than all other implementations, including the ones not providing FIFO. CAFÉ continues demonstrating high throughput even when the number of threads increases up to 64.

As we can see in Figure 2, the results of both the Michael-Scott concurrent queue and ED pools are lower than those of other algorithms. This differs from the results demonstrated by Afek *et al.* [1], where ED pools were shown to clearly outperform standard Java queues. This discrepancy seems to follow from differences in the hardware architectures used in our experiments. Afek *et al.* use a Sun UltraSPARC T2 machine with 2 processors of 64 hardware threads each, while in our system there are 8 quad-cores. The difference in architecture is

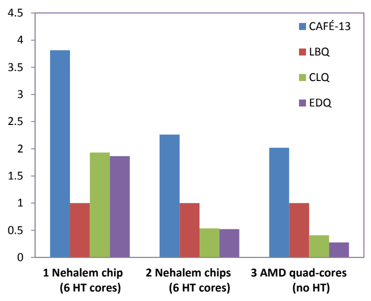


Fig. 3. Throughput on different hardware architectures, normalized by the throughput of LBQ. There are 6 producer threads and 6 consumer threads.

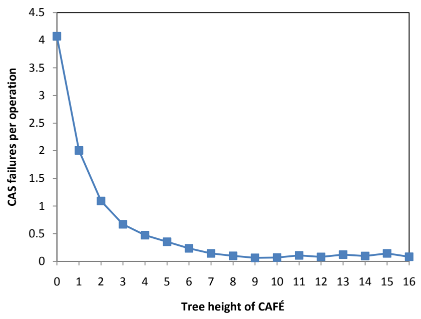
significant due to the *non-uniform* memory access time in multi-processor systems: accessing a memory location from multiple processors is significantly slower than accessing it from multiple hardware threads on the same chip, which usually share a last-level cache. We now show how the non-uniformity of memory accesses influences performance.

Figure 3 demonstrates the throughput of the algorithms in three different configurations: a single Nehalem chip with 6 hyper-thread cores, two Nehalem chips with 6 hyper-thread cores and three AMD quad-cores with no hyper-threading. The algorithms are run with 6 producers and 6 consumers (corresponding to the number of hardware threads available in a single Nehalem chip); the throughput is normalized by the throughput of the Java LinkedBlockingQueue.

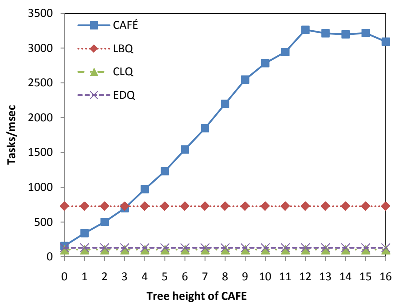
We observe that, consistent with the findings of Afek *et al.*, both ED pools and MS non-blocking queue perform twice as well as Java's linked blocking queue when running on a single chip. However, their performances decrease significantly in systems with two or more chips, when memory sharing becomes more expensive. We point out that CAFÉ continues to outperform all the other algorithms even in the single-chip case, which is the best setting for ED pools and the MS queue. Nevertheless, it is worth mentioning that in [1], ED pools achieved the best results when run on many threads (up to 64) on the same core. We were unable to reproduce these results as we do not have a machine with more than 12 HW threads per chip.

5.3 Choosing the Tree Height

In Figure 4 we demonstrate CAFÉ's performance for 16 producers and 16 consumers as a function of tree height. Figure 4(a) shows the average number of CAS failures per insertion / removal operation. For height $= 0$, CAFÉ is equivalent to the Michael-Scott concurrent queue, and there are 4 CAS failures per operation. The rate of CAS failures drops quickly for larger trees, becoming less than 0.1 for CAFÉ-8.



(a) CAS failures per operation as a function of tree height



(b) CAFÉ throughput as a function of tree height

Fig. 4. CAS failures and system throughput as a function of CAFÉ's tree height for 16 producers and 16 consumers. Small trees induce high contention because of linked list manipulations and reduced tree randomization. Excessively large trees induce contention among producers and consumers operating in the same tree.

The statistics of CAS failures match the throughput graph shown in Figure 4(b). Increasing the tree height improves throughput up to a certain point (12 in our workload), but beyond this performance plateaus. This is because for intermediate tree sizes, producers and consumers usually find themselves in different trees (the latter lagging behind the former), while for heights larger than 13, most of the threads operate in the same tree, which increases contention and decreases performance.

6 Conclusions

We presented CAFÉ, an efficient wait-free task pool with adjustable fairness and contention. CAFÉ uses a scalable TreeContainer building block, which greatly improves on the performance of queue-based alternatives and provides polylogarithmic step complexity for its put operations. Our evaluations show that CAFÉ significantly outperforms both FIFO and non-FIFO task pool algorithms in multi-chip architectures. As we've seen, existing task pools make different trade-offs between fairness and contention. We believe an interesting theoretical question is whether this trade-off is inherent.

References

1. Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 151–162. Springer, Heidelberg (2010)
2. Basin, D., Fan, R., Keidar, I., Kiselov, O., Perelman, D.: Scalable producer-consumer task pools with adjustable fairness and contention. Technical report, Technion, CCIT 790 (2011)
3. Ellen, F., Lev, Y., Luchangco, V., Moir, M.: Snzi: scalable nonzero indicators. In: PODC 2007: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 13–22 (2007)
4. Goodman, J.R., Vernon, M.K., Woest, P.J.: Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-III, pp. 64–75 (1989)
5. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
6. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free fifo queues. Distributed Computing 20, 323–341 (2008)
7. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1996, pp. 267–275 (1996)
8. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2005, pp. 253–262 (2005)

Oblivious Collaboration

Yehuda Afek[1], Yakov Babichenko[2], Uriel Feige[3],
Eli Gafni[4], Nati Linial[5], and Benny Sudakov[6]

[1] The Blavatnik School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel
[2] Department of Mathematics, Hebrew University, Jerusalem, Israel
[3] Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel
[4] Computer Science Department, Univ. of California, Los Angeles, California
[5] School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel
[6] Department of Mathematics, UCLA, Los Angeles, California

Abstract. We introduce *oblivious protocols*, a new framework for distributed computation with limited communication. Within this model we consider the *musical chairs* task $MC(n, m)$, involving n players (processors) and m chairs. Initially, players occupy arbitrary chairs. Two players are in conflict if they both occupy the same chair. The task terminates when there are no conflicts and each player occupies a different chair. Our oblivious protocols use only limited communication, and do so in an asynchronous fashion. Essentially, a player can only observe whether the player itself is in conflict or not, and nothing else. A player observing no conflict halts and never changes its chair, whereas a player observing a conflict changes its chair according to its deterministic program. Known results imply that even with more general communication primitives, no strategy of the players can guarantee termination if $m < 2n-1$. We show that even with this minimal communication termination can be guaranteed with only $m = 2n-1$ chairs. Our oblivious protocol can be extended to the well-known *Adaptive Renaming* problem, using a name-space that is as small as that of the optimal nonoblivious protocol.

We also make substantial progress in optimizing other parameters (such as program length) for our protocols, though many interesting questions remain open.

1 Introduction

In every distributed algorithm each processor must occasionally observe the activities of other processors. This can be realized by explicit communication primitives (such as by reading the messages that other processors send, or by inspecting some publicly accessible memory cell into which they write), or by sensing an effect on the environment due to the actions of other processors. Examples for the latter case are collision detection based algorithms for sharing Multi-Access broadcast media [19]. In our work, in analogy to the collision detection setting, we consider two severe limitations on the processors' behavior and ask how this affects the system's computational power: (i) A processor can only post a proposal for its own output, (ii) Each processor is "blindfolded" and is only occasionally provided with the least possible amount of information, namely a single bit that indicates whether its current state is "good" or "bad". Here "bad/good" stands for whether or not this state conflicts with the global-state desired by the processor. Moreover, we also impose the requirement that algorithms are deterministic (use no randomization). This new minimalist model, properly defined, is called the *oblivious model.* This model might appear to be significantly weaker than other (deterministic) models studied in distributed computing. Yet, we show that two variants of the renaming problem, *adaptive renaming* (AR) (defined in [2]) and *musical chairs* (MC) (introduced in here) can be solved optimally within the oblivious model. Furthermore, we discuss the efficiency of oblivious solutions to these problems and the relations between the oblivious model and the wait-free asynchronous shared memory model with only reads and writes.

The current paper defines the oblivious model in general, but presents results only for the tasks MC and AR, and only with the collision predicate (which is natural for these tasks). We believe that the study of other tasks within the oblivious model can lead to additional interesting insights about the role of communication in distributed computing, though this is left to future work.

The *oblivious* model limits the operations available to individual processors. We find it convenient to model these limitations via a fictitious oracle. Associated with every state of a participating processor is a proposed output, though there could be several different states with the same proposed output. The state at which a processor halts thus defines its final output. The only way a processor can sense its environment is by querying the oracle about a single predicate on the current vector of outputs of the processors. Based on the single bit answer the processor needs to either halt with its current output, or proceed with its computation and propose a new output. But how can a processor's computation proceed? It has no information about the state of other processors (beyond the one bit that tells it that it must proceed), and we are forbidding randomization. Consequently, a processor's proposed output can depend only on its current state, and therefore the sequence of states that processor p_i traverses is simply an infinite word π_i over the alphabet of possible outputs. Upon receiving a negative answer from the oracle, processor p_i in state $\pi_i[k]$ moves to state $\pi_i[k+1]$. Given the definition of a computational task, it is up to the programmer to design the words π_i and the query that each processor poses to the oracle under which that task is always realized properly. Our only assumption is that the oracle correctly answers the queries, and a processor eventually halts/proceeds to the next state in his word upon a bad/good response from the oracle.

The *Musical Chairs*, $MC(n, m)$ task involves n *processors* $p_1, \ldots, p_n$ and, m *chairs* numbered $1, \ldots, m$. Each processor p_i starts in an arbitrary chair, dictated by the input. If the input chairs are all different, all processors are good and the input is the output. Otherwise, the task calls for each processor to capture a chair that differs from the chair captured by any other processor.

The *AdaptiveRenaming*(n, m) (AR(n, m)) task is a close relative of $MC(n, m)$. There are m slots (chairs) numbered $1, \ldots, m$ and each participant has to capture a different slot. The processors have no input. If only $k < n$ processors participate, then each has to capture (output) a different slot from the first $\min(2k-1, m)$ slots. If all the n processors participate then each captures a different slot from the m slots.

In Section 2 we define the oblivious model in detail. For the MC and the AR problems we use the *collision* query – a processor is good iff it is the only one to propose the current chair. We show that in this case the general oblivious model simplifies considerably. These simplifications later help us produce an optimal solution. The infinite words (programs) considered here are an infinite repetition of a finite word.

Remarkably, for each processor we produce a program which is a single cyclic word (an infinite repetition of that word) on an alphabet of chairs. Furthermore, for the MC task the program can be started at any location in the word. This provides for self stabilization [11,12]. Namely, consider a system configuration where each processor occupies a different chair and there are no conflicts. Suppose that the system gets perturbed, and program counters change arbitrarily. This may create conflicts, but the system will nevertheless resettle obliviously in finite time into a conflict-free safe configuration.

Here are the main results presented in the current paper:

1. The introduction of the general oblivious model and its specialization to the problems at hand.
2. A proof that there are tasks that are solvable in a wait-free asynchronous shared memory model with only reads and writes, but not solvable obliviously.
3. The characterization of the minimal m for which there is an oblivious $MC(n, m)$ algorithm:

   Theorem 1. *There is an oblivious $MC(n, m)$ algorithm if and only if $m \geq 2n - 1$.*

   Moreover, for all $N > n$ there exist N words on m chairs such that any n out of the N words constitute an oblivious $MC(n, 2n-1)$ algorithm.
4. The characterization of the minimal m for which there is an oblivious AR(n, m) algorithm:

   Theorem 2. *There is an oblivious $AR(n, m)$ algorithm if and only if $m \geq 2n - 1$.*

5. The words in Theorem 1 use the least number of chairs, namely $m = 2n - 1$. However, the length of these words is doubly exponential in n. Are there oblivious MC algorithms with much shorter words? Even length $O(n)$? Perhaps even length m? How long can the scheduler survive? Here we consider systems with $N \geq n$ words (programs) and any n out of the N should constitute a solution of MC. We call these $MC(n, m)$ systems with N words.

Theorem 3. *For every $N \geq n$, almost every choice of N random words of length $cn \log N$ in an alphabet of $m = 7n$ letters is an $MC(n, m)$ system with N* full *words (words that contain every letter in $1, \ldots, m$). Moreover, every schedule on these words terminates in $O(n \log N)$ steps. Here c is an absolute constant.*

6. Since we are dealing with full words (words that contain every letter in $1, \ldots, m$) and we seek to make them short, we are ultimately led to consider the case where each word is a permutation on $[m]$. At the moment the main reason to study this question is its aesthetic appeal. We can design permutation-based oblivious $MC(n, 2n - 1)$ algorithms for very small n (provably for $n = 3$, computer assisted proof for $n = 4$). We suspect that no such constructions are possible for large values of n, but we are unable at present to show this. We do know, though that

   Theorem 4. *For every integer $d \geq 1$ there is a collection of $N = n^d$ permutations on $m = cn$ symbols such that every n of these permutations constitute an oblivious $MC(n, m)$ algorithm. The constant c depends only on d. In fact, this holds for almost every choice of N random permutations on $[m]$.*

   We should stress that our proofs of Theorems 3 and 4 are purely existential. The explicit construction of such systems of words remains largely open, though we do have some results in this direction, e.g.,

   Theorem 5. *For every integer $d \geq 1$ there is an explicitly constructed collection of $N = n^d$ permutations on $m = O(d^2 n^2)$ symbols such that every n of these permutations constitute an oblivious $MC(n, m)$ algorithm.*

1.1 Related Work

Two variations of the renaming problem were introduced in [2], *Weak Renaming* and *Adaptive Renaming* (AR). In the former, there is an unbounded universe of processor ids of which n wake up and have to each select a different name in the range $\{1, \ldots M(n)\}$. In the AR$(n, M(n))$ problem, which is one of the two problems solved obliviously in this paper, the universe consists of n processors, $\{p_1, \ldots, p_n\}$ and again they have to each capture a different integer in the range $\{1, \ldots M(n)\}$. Yet in AR, if the size of the participating set is $k < n$, outputs are restricted to be in $\{1, \ldots, 2k - 1\}$ (hence the algorithm is adaptive to the number of processors participating). The renaming algorithm presented in [2] solves both variants with $M(n) = 2n - 1$.

Weak renaming is solvable with $M(n) = 2n-2$ for infinitely many different n's, called "exceptional" [10]. For AR$(n, M(n))$, $M(n) = 2n - 1$ is a lower bound as shown in [15]. The proof of this lower bound builds upon previous impossibility results for *set consensus* (see [9,18,20]), by showing that given a hypothetical algorithm for AR$(n, 2n - 2)$ in addition to read-write registers one can solve set consensus in a wait-free manner.

Both weak and adaptive renaming algorithms have been extensively studied over the last two and a half decades, but aside from the above mentioned works concerning solvability, all the studies are about complexity, which is not the subject of this paper.

The musical chairs problem is weakly related to the Musical Benches (MB) [16] problem. In MB there are n benches and $n + 1$ players. Each bench has two seats. Every player needs to occupy a seat, and more than one player can occupy the same seat. An output is legal if in every bench at most one seat is occupied. Initially, players occupy arbitrary seats. If the initial configuration is legal, it has to be the output. However, if the initial configuration is not legal, then players can move and must return a legal output. In [16] it is shown, using the Bursuk-Ulam Theorem, that the task for $n = 2$ has no wait-free solution in an asynchronous shared memory model with only reads and writes. MC shares with MB the flavor of the game of "jumping" from seat to seat. However, MC is about separating players from each other, whereas MB is about getting players on a bench to arrive at consensus.

There are other contexts in which algorithms of an oblivious nature were considered. An algorithm in which each process is assigned a permutation which specifies the order it is to do work is presented in [3]. Algorithms to compute a maximal independent set with only carrier sensing with or without collision detection are provided in this volume [6]. Among all work on algorithms with an oblivious nature, we find it most instructive to compare our work with work on *universal traversal sequences* (UTS) for covering graphs. A word over the alphabet $\{0, 1, \ldots d - 1\}$ can guide a walk on an n-vertex d-regular undirected graph: in each step the walk selects its next out-going edge according to the respective symbol of the word. Such a word is a UTS if for *every* connected n-vertex d-regular graph, regardless of how each vertex labels its out-going edge, the corresponding walk visits all vertices of the graph. In [1] it is shown that a sufficiently long random word (say, of length n^5) is almost surely a UTS. In analogy, our proof of Theorem 3 shows that sufficiently long random words almost surely form MC algorithms. However, in our case the proof needs to overcome an obstacle not present in the UTS case. The difference is that in MC, as words get longer, the scheduler also gets more choices of how to schedule them, whereas for UTS the number of graphs is fixed independently of the length of the words. As a consequence, for some range of parameters (e.g., provably when $m < 2n - 1$, as Theorem 1 shows), the statement is simply not true. There are no analogous forbidden ranges of parameters for universal traversal sequences.

1.2 Discussion

Due to space limitations, large parts (including most proofs) are omitted from the current version of this paper. The reader interested in more details is referred to [5].

A number of simple observations follow from the requirement that oblivious algorithms are deterministic. (i) An oblivious $MC(n, m)$ algorithm cannot include any two identical words. Otherwise the corresponding players might move together in lock-step, constantly being in collision. Hence it is essential that no two processors have the same program. (ii) For every oblivious $MC(n, m)$ algorithm with cyclic words, there is a finite upper bound on the number of moves a processor can make before termination. This is because there are only finitely many system configurations (a system configuration is determined by one state for each processor, and the number of possible states of a processor is equal to the length of the cycle in its cyclic word, and hence finite), and in a terminating sequence of moves no system configuration can be visited twice. (iii) In fact, for every collection of finite words there is a directed graph whose vertices are all the system configurations. Edges correspond to the possible transitions. The collection of words constitute an oblivious MC protocol iff this graph is acyclic.

Not all aspects of oblivious protocols are required for the purpose of the lower bound $m \geq 2n - 1$. The two crucial aspects are the asynchrony of the model, and the fact that our algorithms are deterministic (no randomization). In a synchronous setting (where in every time step, every processor involved in a collision moves to its next state), $m = n$ suffices even for oblivious protocols. (This can be proven using the techniques of Theorem 3. Details are in [5].) Likewise, $m = n$ suffice if randomization is allowed – with probability 1 eventually there are no collisions. However, no specific upper bound on the number of steps can be guaranteed in this case. Moreover, if the randomized algorithms are run using pseudorandom generators (rather than true randomness) the argument breaks. For any fixed seed of a pseudorandom generator, the algorithm becomes deterministic and the lower bound $m \geq 2n - 1$ holds.

The lower bound of $m \geq 2n - 1$ uses some benign-looking aspects of the MC task, so further discussion is called for. Recall that each processor starts in an arbitrary chair, dictated by the input. In the absence of an external input specifying the starting chair, a trivial oblivious MC algorithm (with $m = n$) contains n distinct single-letter words. Another requirement is that if the input chairs are all different, all processors are good and the input is the output. Without such a requirement, the processors might simply ignore the initial input and the trivial oblivious MC algorithm would still apply. Hence the lower bound of $m \geq 2n - 1$ depends on requirements beyond the need for each processor to capture a different chair. Here this extra requirement is the possibility to dictate an output. This particular requirement makes it easy to transfer previously existing lower bounds to our MC problem.

Our present proof for the lower bound of $m \geq 2n - 1$ leaves something to be desired. It relies on previous nontrivial work in distributed computing. What's worse is that we prove a lower bound for a simple oblivious model via a reduction to a lower bound proved in a more complicated model. This roundabout approach obscures the essential properties that make the lower bound work. Indeed, in a companion manuscript (in preparation), we present a self contained proof for the lower bound of $m \geq 2n - 1$. That presentation clarifies the minimal requirements that are needed in order to make the lower bound work. In particular, it is not necessary that one can dictate an arbitrary starting chair for each processor – dictating one of two chairs suffices.

As noted, we design oblivious $MC(n,m)$ protocols with $m = 2n - 1$. Part of our work also concerns analyzing what ratios between m and n one can obtain using collections of randomly chosen words as in Theorem [3]. As explained in the introduction, this allows us to present more efficient deterministic oblivious programs – though random words seem to need more chairs, they can reach conflict free configurations more quickly. Moreover, the use of random words is a design principle that can be applied to design oblivious algorithms for other tasks as well. Developing an understanding of what they can achieve and techniques for their analysis is likely to pay off in the long run. One of the major questions that remain open in our work is whether randomly chosen words can be used to design deterministic oblivious MC protocols with $m = 2n - 1$.

2 The Oblivious Model

The *Oblivious* model (formally defined in [5]) is an asynchronous distributed computing model in which each processor, at each point of time, exposes an output value it currently proposes, and may receive at most one bit of information. This bit indicates whether its proposed output is legal with respect to the other currently proposed outputs (and hence the processor may halt) or not (and then the processor should continue the computation). If a processor decides to halt at the current state, then its proposed output is its final output. We denote the set of possible output values by O. A *system configuration* (or configuration for short) is a vector of n elements, one per processor, whose entries come from the set $O \cup \{\perp\}$. Here $\perp$ represents a processor that has not yet proposed any output, either because it is not participating, or because it was not scheduled yet to propose an output (these two cases are indistinguishable to other processors). An entry from O represents the output a corresponding processor proposes in the configuration. In an oblivious algorithm correctly designed for a given task, eventually all participating processors must halt, and the final configuration must be a legal output vector in the task specification.

The defining feature of the *oblivious* model is that each processor may receive only one bit of information about the system configuration in each computation step. Namely, for each processor there is a predicate that maps configurations to one of two values, one dictating that the processor will change its state, the other dictating that it should halt in its current state. In the most general setting the predicate provided for each processor may depend on its input. However, throughout an execution one predicate is used for each processor. A necessary (but not sufficient) condition for correct oblivious algorithms is that in every illegal configuration at least one processor's predicate dictates a change of state. Our formal model does not exclude the use of arbitrary complex predicates (as long as they depend only on the current configuration), but oblivious algorithms have greater appeal when the predicates involved are simple and natural. For the two tasks considered in this paper, the same *collision* predicate is used by all the processors.

Initially, and as a function of its input, each processor p_i selects a word π_i over O, and a predicate $pred_i$ on the set of of all configurations. The first letter in π_i is p_i's input, i.e., $\pi_i[1] = input_i \in O$. For tasks such as AR in which a processor need not have any input, the first letter is set to be an output that is valid if no other processor participates (hence for AR the first letter is 1).

We describe the system using the notion of an omnipotent know-all scheduler called *asynchronous* (other schedulers with different names are described in the sequel). Execution under the control of the asynchronous scheduler proceeds in rounds. The scheduler maintains a set P of participating processors, a set $E \subset P$ of enabled processors, and a set $DONE$ (disjoint from P) of processors that have already halted. These sets are initially empty. In each round the scheduler performs the following sequence of operations. It may add some not yet participating processors to P. It may evaluate the predicate $pred_i$ for some subset of processors in $P \setminus E$. If $pred_i$ evaluates to true, the scheduler adds processor p_i to the set E. Otherwise, if it evaluates to false, it removes p_i from P and adds it to the set $DONE$. Finally, the scheduler selects a subset $SE \subseteq E$, removes it from E, and moves each $p_i \in SE$ to its next letter in π_i. I.e., the current output of p_i is replaced by the next one in its program, π_i. This completes the round.

An oblivious algorithm solves a task if for every input vector, the scheduler is forced to eventually place all participating processors in the $DONE$ set. At that point it can no longer continue, and the final configuration is such that $(v_{inp}, v_{out}) \in \Delta$, the relation that defines the task.

The asynchronous scheduler for oblivious algorithms mimics the behavior of a wait-free algorithm in an asynchronous shared memory model with only reads and writes, on configurations. Theorem 6 below is proved in [5] simply by having each processor emulate the scheduler through reads (snapshots) and writes of its newly proposed output in shared memory.

Theorem 6. *Every task that is solvable obliviously has a wait-free solution in an asynchronous shared memory model with only reads and writes.*

Thus the oblivious model is subsumed by the wait-free asynchronous shared memory model with only reads and writes. Is this a proper inclusion? To clarify the answer we introduce an intermediate class of tasks that we call Output Negotiation, or ON. It includes those tasks that have a wait-free solution in an asynchronous shared memory model with only reads and writes in a system where writing is in the oblivious model (processors can only expose their proposed outputs), whereas reading is as in the general wait-free asynchronous shared memory model with only reads and writes (a processor can read all exposed information rather than only a single predicate). By definition, every obliviously solvable task is ON solvable.

Corollary 7. *Every obliviously solvable task is in ON.*

Obviously, ON is a subset of the wait-free asynchronous shared memory model with only reads and writes, and in Theorem 8 below whose proof is in [5] we show that this inclusion is proper. In the proof we consider the task *AntiMC* which is a variation on epsilon agreement [13] and show that AntiMC is not solvable just by communicating outputs. AntiMC is a task with 3 processors whose input and output are each a number in $\{1, \dots, 5\}$. A processor running alone must output its input. If more than one processor participates, all the outputs must lie within two consecutive numbers.

Theorem 8. *There exists a task, AntiMC, that has a wait-free solution in an asynchronous shared memory model with only reads and writes but does not belong to ON.*

2.1 Impossibility of $MC(n, 2n-2)$

In Sections 3 and 4 we show that $MC(n, 2n-1)$ and AR$(n, 2n-1)$ are solvable obliviously. AR$(n, 2n-2)$ has no wait-free solution in an asynchronous shared memory model with only reads and writes [14,15], and hence not solvable obliviously either. Theorem 9 whose proof is in [5] shows a reduction from AR$(n, 2n-2)$ to $MC(n, 2n-2)$. This implies that $MC(n, 2n-2)$ has no wait-free solution in an asynchronous shared memory model with only reads and writes, and hence also not solvable obliviously.

Theorem 9. *$AR(n, 2n-2)$ is wait-free reducible to $MC(n, 2n-2)$ in an asynchronous shared memory model with only reads and writes.*

2.2 Cyclic Finite Programs or Words

For the AR task, processors have no input (or alternatively, are assumed to always have the input 1), and hence each processor has only one sequence. Our constructions of oblivious algorithms all have the property that the same sequence is used for all inputs. Moreover, we consider finite sequences over which the processor goes cyclically. In the MC task one can designate m locations in the word, each corresponding to a possible output that has been dictated by the input to the processor and each processor advances cyclically on the word starting from that designated location.

2.3 Simplified Oblivious Model for MC and AR

The use of the collision predicate can be shown to imply that for the AR and MC problems it is sufficient to consider a much simpler scheduler, the ***Pairwise Immediate scheduler***: In each round this scheduler selects two processors that are currently colliding with each other, and moves either one or both of them, c'est tout. Suppose that every processor has an associated word. We show that given an initial configuration (starting positions on the words), the oblivious asynchronous scheduler runs to infinity iff the pairwise immediate scheduler does. This scheduler is then used in constructions in Sections 3 and 4. The constructions in Section 5 use an even more restrictive scheduler, the ***Canonical scheduler***. Like the pairwise immediate scheduler, the canonical scheduler can move only one or both of two currently colliding processors, but unlike the pairwise immediate scheduler, the choice of which two colliding processors to consider is not made by the scheduler, but rather dictated to it. For formal definitions of these schedulers and the proof of their equivalence, see [5].

3 An Oblivious MC Algorithm with $2n - 1$ Chairs

This section is dedicated to the upper bound that is stated in Theorem [1]. The proof of this theorem is inductive and rather technical. For lack of space, the full text with all proofs is given in [5]. In what follows we attempt to give the reader a sense of the main ingredients of the construction and how they come together in the proof.

3.1 Preliminaries

The length of a word w is denoted by $|w|$. The concatenation of words is denoted by $\circ$. The r-th power of w is denoted by $w^r = w \circ w \ldots \circ w$ (r times). Given a word π and a letter c, we denote by $c \otimes \pi$ the word in which the letters are alternately c and a letter from π in consecutive order. For example if $\pi = 2343$ and $c = 1$ then $c \otimes \pi = 12131413$. A collection of words $\pi_1, \pi_2, ..., \pi_n$ is called *terminal* if no schedule can fully traverse even one of the π_i. Note that we can construct a terminal collection from any MC algorithm just by raising each word to a high enough power.

We now introduce some of our basic machinery in this area. A key tool is a method to extend terminal sets of words.

Proposition 1. *Let n, m, N be integers with $1 < n < m$. Let $\Pi = \{\pi_1, \ldots \pi_N\}$ be a collection of m-full words such that*

$$\text{every } n \text{ of these words form an oblivious } MC(n, m) \text{ algorithm.} \tag{1}$$

Then Π can be extended to a set of $N+1$ m-full words that satisfy condition (1).

proof. Suppose that for every choice of n words from Π and for every initial configuration no schedule lasts more than t steps. (By the pigeonhole principle $t \leq L^n$, where L is the length of the longest word in Π). For a word π, let π' be defined as follows: If $|\pi| \geq t$, then $\pi' = \pi$. Otherwise it consists of the first t letters in π^r where $r > |\pi|/t$. The new word that we introduce is $\pi_{N+1} = \pi'_1 \circ \pi'_2 \circ \ldots \circ \pi'_n$. It is a full word, since it contains the full word π_1 as a sub-word.

We need to show that every set Π' of $n-1$ words from Π together with π_{N+1} constitute an oblivious $MC(n, m)$ algorithm. Observe that in any infinite schedule involving these words, the word π_{N+1} must move infinitely often. Otherwise, if it remains on a letter c from some point on, replace the word π_{N+1} by an arbitrary word from $\Pi - \Pi'$ and stay put on the letter c in this word. This contradicts our assumption concerning Π. (Note that this word contains the letter c by our fullness assumption.) But π_{N+1} moves infinitely often, and it is a concatenation of n words whereas Π' contains only $n-1$ words. Therefore eventually π_{N+1} must reach the beginning of a word π_α for some $\pi_\alpha \notin \Pi'$. From this point onward, π_{N+1} cannot proceed for t additional steps, contrary to our assumption. □

Note that by repeated application of Proposition 1, we can construct an arbitrarily large collection of m-full words that satisfy condition (1).

We next deal with the following situation: Suppose that $\pi_1, \pi_2, ..., \pi_m$ is a terminal collection, and we concatenate an arbitrary word σ to one of the words π_i. We show that by raising all words to a high enough power we again have a terminal collection in our hands.

Lemma 1. *Let $\pi_1, \pi_2, ..., \pi_p$ be a terminal collection of full words over some alphabet. Let σ be an arbitrary full word over the same alphabet. Then the collection*

$$(\pi_1)^k, (\pi_2)^k, ..., (\pi_{i-1})^k, (\pi_i \circ \sigma)^2, (\pi_{i+1})^k, ..., (\pi_p)^k$$

is terminal as well, for every $1 \le i \le p$, and every $k \ge |\pi_i| + |\sigma|$.

Lemma 1 yields immediately:

Corollary 10. *Let $\pi_1, \pi_2, ..., \pi_p$ be a terminal collection of full words over some alphabet, and let $\pi_{p+1}, \pi_{p+2}, ..., \pi_n$ be arbitrary full words over the same alphabet. Then the collection*

$$(\pi_1 \circ \pi_2 \circ ... \circ \pi_n)^2, (\pi_1)^k, (\pi_2)^k, ..., (\pi_{i-1})^k, (\pi_{i+1})^k, ..., (\pi_p)^k$$

is terminal as well. This holds for every $1 \le i \le p$ and $k \ge \sum_{i=1}^{n} |\pi_i|$.

3.2 The MC$(n, 2n-1)$ Upper Bound

Our proof shows somewhat more than Theorem 1 says (see Proposition 2). We do this, since the scheduler can "trade" a player P for a chair c. Namely, he can keep P constantly on chair c. This allows the scheduler to move any other player past c-chairs. In other words this effectively means the elimination of chair c from all other words. This suggests the following definition: If π is a word over alphabet C and $B \subseteq C$, we denote by $\pi(B)$ the word obtained from π by deleting from it the letters from $C \setminus B$.

Our construction is recursive. An inductive step should add one player (i.e., a word) and two chairs. We carry out this step in two installments: In the first we add a single chair and in the second one we add a chair and a player. Both steps are accompanied by conditions that counter the above-mentioned trading option.

Proposition 2. *For every integer $n \ge 1$*

- *There exist full words $s_1, s_2, ..., s_n$ over the alphabet $\{1, 2, ..., 2n-1\}$ such that $s_1(A), s_2(A), ..., s_p(A)$ is a terminal collection for every $p \le n$ and for every subset $A \subseteq \{1, 2, ..., 2n-1\}$ of cardinality $|A| = 2p-1$.*
- *There exist full words $w_1, w_2, ..., w_n$ over alphabet $\{1, 2..., 2n\}$, such that $w_1(B), w_2(B), ..., w_p(B)$ is a terminal collection for every $p \le n$ and for every subset $B \subseteq \{1, 2, ..., 2n\}$ of cardinality $|B| = 2p-1$.*

The words $s_1, s_2, ..., s_n$ in Proposition 2 constitute a terminal collection and are hence an oblivious $MC(n, 2n-1)$ algorithm that proves the upper bound part of Theorem 1. The proof of Proposition 2 is given in [5].

4 The Oblivious AR$(n, 2n-1)$ Algorithm

The ideas developed to solve the musical chairs problem and prove Theorem 1 turn out to yield as well an answer to the oblivious AR problem and a proof of Theorem 2. The rules are the same as in the MC problem, except that the scheduler cannot select the initial positions, and every word is started at its first letter. In order to prove Theorem 2 we should construct a collection of full words $\Pi_N = \{s_1, s_2, ..., s_N\}$ over the alphabet $[2N-1]$ such that for every $n \leq N$ and for every set of n words from Π_N the following holds: Every schedule that starts from the first letter in each of these words reaches a safe configuration and all players only visits chairs from the set $\{1, \ldots, 2n-1\}$.

We note that our construction yields very long words - triply exponential in N. It is an interesting challenge to accomplish this with substantially shorter words.

proof (Theorem 2). By Proposition 1 and Theorem 1, we can construct for each $1 \leq i, n \leq N$ a word $\pi_{i,n}$ that is $[2n-1]$-full such that every set of n words in the set $\{\pi_{i,n} | i = 1, \ldots, N\}$ constitute an oblivious $MC(n, 2n-1)$ protocol.

We show that with a proper choice of the exponents $l_1, \ldots, l_N$, the Theorem holds with the words $s_i = \pi_{i,1}^{l_1} \circ \pi_{i,2}^{l_2} \circ \ldots \circ \pi_{i,N}^{l_N}$.

The theorem follows if we can show that for every $1 \leq n \leq N$ and every subset $J \subseteq [N]$ of cardinality $|J| = n$ the following holds: In every possible schedule that starts each word in $\{s_j | j \in J\}$ from its first letter, no player reaches a position beyond the subword $\pi_{j,n}^{l_n}$. Consider any point in such a schedule. Say that player P_j (for some $j \in J$) is *leading* if it currently resides in the stretch $\pi_{j,n}^{l_n}$ of s_j. Otherwise, we say that j is *trailing*. We observe that during a period of time in which no trailing player changes position, no leading player can traverse a complete copy of $\pi_{j,n}$. To see this, consider an arbitrary MC schedule with the words $\{\pi_{j,n} | j \in J\}$. We start this schedule as follows: Every leading player maintains his position from the original AR schedule and every trailing player stays put on the same chair that he is currently occupying. (Such a chair can be found in the word $\pi_{j,n}$ since it is $[2n-1]$-full). The claim follows since the words $\{\pi_{j,n} | j \in J\}$ constitute an oblivious $MC(n, 2n-1)$ protocol.

It follows that no leading player P_j can traverse more than $\sum_{\nu < n, i \in J \setminus \{j\}} |\pi_{i,\nu}| l_\nu$ copies of $\pi_{j,n}$ in s_j. Our claim follows if we choose l_j that is larger than this integer. □

5 Oblivious MC Algorithms by the Probabilistic Method

Remark 11. *It is important to note that the protocols that are presented in this section are* deterministic*. The constructions are, however,* inexplicit *and the existence of good protocols is proved by using a probabilistic argument. It is an intriguing open problem to find equally good explicit constructions.*

For lack of space, the full text with all proofs is given in [5]. In what follows we attempt to sketch the results.

Theorem 3 can be thought of as a (nonconstructive) derandomization of the randomized MC algorithm in which players choose their next chair at random (and future random decisions of players are not accessible to the scheduler). Standard techniques for derandomizing random processes involve taking a union bound over all possible bad events, which in our case corresponds to a union bound over all possible schedules. The asynchronous scheduler has too many options (and so does the immediate scheduler), making a union bound too wasteful. For this reason, in the analysis of this protocol we consider the canonical scheduler, which is as powerful as the asynchronous scheduler (see Section 2.3). In every unsafe configuration, a *canonical pair* of players in conflict is dictated to the canonical scheduler, and the canonical scheduler has only three possible moves to choose from. This makes it viable to use a union bound.

In this construction each of the N words is chosen independently at random as a sequence of L chairs, where each chair in the sequence is chosen independently at random. Our proof shows that with high probability (probability tending to 1 as the value of the constant c grows), this choice satisfies Theorem 3.

A simple union bound shows that in this random construction, with high probability, all words are full. Proving termination is more of a challenge. We keep track of all possible schedules. To this end we use “a logbook” that is the complete ternary tree $\mathcal{T}$ of depth L rooted at r. Associated with every node v of $\mathcal{T}$ is a random variable X_v. The values taken by X_v are system configurations. For a given choice of words and an initial system configuration we define the value of X_r to be the chosen initial configuration. Every node v has three children corresponding to the three possible next configurations that are available to the canonical scheduler at configuration X_v (and to an “empty” configuration if the scheduler cannot move). The proof uses a *potential* function that maps a configuration with i occupied chairs to x^{n-i}, where $x > 1$ is a constant optimized within the proof. In a nonempty configuration the potential is at least 1. Associated with every node of $\mathcal{T}$ is a nonnegative random variable $P = P_v$ that is the potential of the (random) configuration X_v. The main step of the proof is to show that if v_1, v_2, v_3 are the three children of v, then $\sum_{i=1}^{3} \mathbb{E}(P_{v_i}) \leq r\mathbb{E}(P_v)$ for some constant $r \leq 0.99$. This exponential drop implies that

$$\mathbb{E}\Big(\sum_{v \text{ is a leaf of } \mathcal{T}} (P_v)\Big) = \sum_{v \text{ is a leaf of } \mathcal{T}} \mathbb{E}(P_v) = o(1)$$

provided that L is large enough. This implies that with probability $1 - o(1)$ (over the choice of random words) all leaves of $\mathcal{T}$ correspond to an empty configuration. In other words every schedule terminates in fewer than L steps.

5.1 Permutations over $O(n)$ Chairs

The argument that proves Theorem 3 is inappropriate for the proof of Theorem 4. Theorem 4 deals with random permutations, whereas in the proof of Theorem 3 we use words of length $\Omega(n \log n)$. (Longer words are crucial there for two main reasons: To guarantee that words are full and to avoid wrap-around. The latter property is needed to guarantee independence.) Indeed in proving Theorem 4 our arguments are substantially different. In particular, we work with a pairwise immediate scheduler, and unlike the proof of Theorem 3, there does not appear to be any significant benefit (e.g., no significant reduction in the ratio $\frac{m}{n}$) if a canonical scheduler is used instead.

Here are some of the main ingredients of the proof of Theorem 4 for the special case $N = n$ (a slight extension of these ideas proves the general case). We show that with high probability, a set of random permutations $\pi_1, \ldots, \pi_n$ has the property that in every possible schedule the players visit at most $L = O(m \log m)$ chairs. Our analysis uses the approach of deferring random decisions until they are actually needed. For each of the m^n possible initial configuration, we consider all possible sequences of L locations. For each such sequence we fill in the chairs in the locations in the sequence at random, and prove that the probability that this sequence represents a possible schedule is extremely small – so small that even if we take a union bound over all initial configurations and over all sequences of length L, we are left with a probability much smaller than 1.

The main difficulty in the proof is that since $L \gg m$ some players may completely traverse their permutation (even more than once) and therefore the chairs in these locations are no longer random. To address this, we partition the sequence of moves into L/t blocks, where in each block players visit a total of $t = \delta m$ locations for some sufficiently small constant δ. Also $n = \epsilon m$, where ϵ is a constant much smaller than δ. This choice of parameters implies that within a block, chairs are essentially random and independent. To deal with dependencies among different blocks, we classify players (and their corresponding permutations) as *light* or *heavy.* A player is *light* if during the whole schedule (of length L) it visits at most $t/\log m = o(t)$ locations. A player that visits more than $t/\log m$ locations during the whole sequence is *heavy.* For light players, the probability of encountering a particular chair in some given location is at most $\frac{1}{m-o(t)} \leq \frac{1+o(1)}{m}$. Hence, the chairs encountered by light players are essentially random and independent (up to negligible error terms). Thus it is the heavy players that introduce dependencies among blocks. Every heavy player visits at least $t/\log m$ locations. Hence the number n_h of heavy players does not exceed $(L \log m)/t = O(\log^2 m)$. The fact that the number of heavy players is small is used in our proof to limit the dependencies among blocks.

6 Open Problems

Our MC algorithms involve very long words. An interesting question is to find explicit constructions with $m = 2n - 1$ chairs and substantially shorter words.

In other ranges of the problem we can show, using the probabilistic method, that oblivious $MC(n, m)$ algorithms exist with $m = O(n)$ and relatively short full words. We still do not have explicit constructions of such protocols. We would also like to determine $\liminf \frac{m}{n}$ such that n random words over an m letter alphabet tend to constitute an oblivious $MC(n, m)$ algorithm.

Computer simulations strongly suggest that for random permutations, a value of $m = 2n - 1$ does not suffice. On the other hand, we have constructed (details omitted from this manuscript) oblivious $MC(n, 2n - 1)$ algorithms using permutations for $n = 3$ and $n = 4$ (for the latter the proof of correctness is computer-assisted). For $n \geq 5$ we have neither been able to find such systems (not even in a fairly extensive computer search) nor to rule out their existence.

A self contained proof of the $m \geq 2n - 1$ lower bound will appear in a subsequent paper. The following question remains open: What is the smallest m for which there are collections of $N = m + 1$ (not necessarily full) words such that every $\min[n, N]$ of them form an oblivious MC algorithm *when starting at the initial chair of each word*. Our proof that $m \geq 2n - 1$ assumes that the scheduler is allowed to pick an arbitrary initial state on each word.

We do not know how hard it is to recognize whether a given collection of words constitute an oblivious MC algorithm. This can be viewed as the problem whether some digraph contains a directed cycle or not. The point is that the digraph is presented in a very compact form. It is not hard to place this problem in PSPACE, but is it in a lower complexity class, such as co-NP or P?

There are interesting foundational questions related to different models in distributed computing. We have defined here the Output Negotiation (ON) model, and showed that it is properly included in the read/write model. It follows by definition that the oblivious model is included in the ON model. It would be interesting to know whether this last inclusion is proper.

Acknowledgements. Work of Uriel Feige was supported in part by The Israel Science Foundation (grant No. 873/08). Work of Benny Sudakov was supported in part by NSF grant DMS-1101185, NSF CAREER award DMS-0812005 and by USA-Israeli BSF grant.

References

1. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovasz, L., Rackoff, C.: Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems. In: FOCS, pp. 218–223 (1979)
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. Anderson, R.J., Woll, H.: Algorithms for the Certified Write-All Problem. SIAM J. Comput. 26(5), 1277–1283 (1997)
4. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM 40(4), 873–890 (1993)
5. Afek, Y., Babichenko, Y., Feige, U., Gafni, E., Linial, N., Sudakov, B.: Oblivious Collaboration (ArXiv version of current paper.), http://arxiv.org/abs/1106.2065
6. Afek, Y., Alon, N., Bar-Joseph, Z., Cornejo, A., Haeupler, B., Kuhn, F.: Beeping a Maximal Independent Set. In: Proc. 25th Int'l Symposium on Distributed Computing (DISC 2011), Rome Italy (September 20-22, 2011)

7. Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., Travers, C.: Simultaneous consensus tasks: A tighter characterization of set-consensus. In: Chaudhuri, S., Das, S.R., Paul, H.S., Tirthapura, S. (eds.) ICDCN 2006. LNCS, vol. 4308, pp. 331–341. Springer, Heidelberg (2006)
8. Beame, P., Blais, E., Ngoc, D.: Longest common subsequences in sets of permutations, `http://arxiv4.library.cornell.edu/abs/0904.1615?context=math`
9. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on Theory of Computing (STOC 1993), pp. 91–100 (1993)
10. Castañeda, A., Rajsbaum, S.: New combinatorial topology upper and lower bounds for renaming. In: PODC, pp. 295–304 (2008)
11. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
12. Dolev, S.: Self-Stabilization. MIT Press, Cambridge, ISBN 0-262-04178-2
13. Dolev, D., Lynch, N.A., Pinter, S., Stark, E.W., Weihl, W.E.: Reaching Approximate Agreement in the Presence of Faults. In: Symposium on Reliability in Distributed Software and Database Systems, pp. 145–154 (1983)
14. Gafni, E.: Read-write reductions. In: Chaudhuri, S., Das, S.R., Paul, H.S., Tirthapura, S. (eds.) ICDCN 2006. LNCS, vol. 4308, pp. 349–354. Springer, Heidelberg (2006)
15. Gafni, E., Mostéfaoui, A., Raynal, M., Travers, C.: From adaptive renaming to set agreement. Theor. Comput. Sci. 410(14), 1328–1335 (2009)
16. Gafni, E., Rajsbaum, S.: Musical benches. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 63–77. Springer, Heidelberg (2005)
17. Gafni, E., Rajsbaum, R., Raynal, M., Travers, C.: The committee decision problem. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 502–514. Springer, Heidelberg (2006)
18. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
19. Metcalfe, R.M., Boggs, D.R.: Ethernet: Distributed packet switching for local computer networks. Commun. Ass. Comput. Mach. 19(7), 395–404 (1976)
20. Saks, M., Zaharoglou, F.: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)

Author Index

Abraham, Ittai 404
Afek, Yehuda 16, 32, 489
Alistarh, Dan 97
Alon, Noga 32
Aspnes, James 97, 363
Attiya, Hagit 374

Babichenko, Yakov 489
Balakrishnan, Mahesh 208
Bansal, Piyush 389
Barenboim, Leonid 66, 237
Bar-Joseph, Ziv 32
Basin, Dmitry 475
Borran, Fatemeh 374
Bouzid, Zohir 330
Boyle, Elette 181
Byshevsky, Anton 125

Chalopin, Jérémie 432
Christoforou, Evgenia 147
Clementi, Andrea E.F. 298
Collins, Andrew 447
Cornejo, Alejandro 32
Czygrinow, Andrzej 200
Czyzowicz, Jurek 447

Das, Shantanu 432
Delporte-Gallet, Carole 145
Dolev, Shlomi 237, 252
Dubois, Swan 150

Elkin, Michael 66

Fan, Rui 475
Farach-Colton, Martín 202
Fauconnier, Hugues 145
Feige, Uriel 489
Fernández Anta, Antonio 147, 202, 225
Fraigniaud, Pierre 333
Freedman, Michael J. 205

Gafni, Eli 145, 489
Gąsieniec, Leszek 447
Gavoille, Cyril 404
Georgiou, Chryssis 147, 165
Ghaffari, Mohsen 268
Gilbert, Seth 252
Goldwasser, Shafi 181
Gopal, Prasant 389
Gradinariu Potop-Butucaru, Maria 82
Gupta, Anuj 389

Haeupler, Bernhard 32
Hakimi, Michael 16
Hanćkowiak, Michal 200
Hutle, Martin 374

Ihm, Sunghwan 205
Izumi, Taisuke 82, 330

Keidar, Idit 125, 475
Khabbazian, Majid 252
Kiselov, Ofer 475
Kosowski, Adrian 447
Kowalski, Dariusz R. 165
Krzywdziński, Krzysztof 200
Kuhn, Fabian 32, 416
Kutten, Shay 197
Kuznetsov, Petr 145

Labourel, Arnaud 432
Lamport, Leslie 141, 211
Larrea, Mikel 328
Lavi, Ron 197
Lewko, Allison 348
Linial, Nati 489
Litmanovich, Oleg 125
Lu, Li 460
Lynch, Nancy 268

Malkhi, Dahlia 208
Markou, Euripides 432
Martin, Russell 447
Marzullo, Keith 143, 326
Masuzawa, Toshimitsu 150
Mei, Alessandro 143
Meling, Hein 143
Milani, Alessia 202
Milosevic, Zarko 374
Morrison, Adam 16
Mosteiro, Miguel A. 147, 202, 225

Newport, Calvin 252
Niranjan Mysore, Radhika 326

Oshman, Rotem 416
Ostrovsky, Rafail 237
Ousterhout, Kay 205

Pelc, Andrzej 1
Perelman, Dmitri 125, 475
Pignolet, Yvonne Anne 311
Prabhakaran, Vijayan 208

Rajsbaum, Sergio 333
Ramón Muñoz, Jorge 225
Raynal, Michel 328

Sastry, Srikanth 268
Schiper, André 374
Schmid, Stefan 311
Schneider, Johannes 51
Scott, Michael L. 460
Sen, Siddhartha 205
Silvestri, Riccardo 298
Srinathan, Kannan 389
Sudakov, Benny 489
Szymańska, Edyta 200

Taubenfeld, Gadi 110
Tauman Kalai, Yael 181
Tirthapura, Srikanta 283
Tixeuil, Sébastien 150, 330
Travers, Corentin 333
Tredan, Gilles 311
Trehan, Amitabh 197

Vahdat, Amin 326
Valero, Mathieu 82
Vasishta, Pranav Kumar 389

Wada, Koichi 330
Walraed-Sullivan, Meg 326
Wattenhofer, Roger 51
Wawrzyniak, Wojciech 200
Widmayer, Peter 332
Wobber, Ted 208
Woodruff, David P. 283

Zaks, Shmuel 202