21st International Conference on Principles of Distributed Systems

OPODIS 2017, December 18–20, 2017, Lisboa, Portugal

Edited by
James Aspnes
Alysson Bessani
Pascal Felber
João Leitão

Editors

James Aspnes
Department of Computer Science
Yale University
james.aspnes@gmail.com

Alysson Bessani
Faculdade de Ciências
Universidade de Lisboa
anbessani@fc.ul.pt

Pascal Felber
Institut d'informatique
Université de Neuchâtel
pascal.felber@unine.ch

João Leitão
Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa
jc.leitao@fct.unl.pt

ACM Classification 1998
C.2.4 Distributed Systems, C.4 Performance of Systems, D.1.3 Concurrent Programming, E.1 Data Structures, F.1.2 Modes of Computation

ISBN 978-3-95977-061-3

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-061-3.

Publication date
March, 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.OPODIS.2017.0

ISBN 978-3-95977-061-3 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Front Matter

Keynotes

Regular Papers

Session 1: Graph and Network Algorithms

Session 2: Concurrency

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS 2017

Session 3: Agents and Robots

Session 4: Shared Memory

Session 5: Algorithms, Randomization and Optimization

Session 6: Replication and Consensus

Session 7: Security and Dependability

Session 8: Distributed Algorithms

Preface

The papers in this volume were presented at the 21st International Conference on Principles of Distributed Systems (OPODIS 2017), held on December 18–20, 2017, in Lisbon, Portugal. The conference was organized by the University of Lisbon and took place at the Faculty of Sciences.

OPODIS is an open forum for the exchange of state-of-the-art knowledge on distributed computing. With strong roots in the theory of distributed systems, OPODIS has expanded its scope to cover the whole range between the theoretical aspects and practical implementations of distributed systems, as well as experimentation and quantitative assessments. All aspects of distributed systems are within the scope of OPODIS: theory, specification, design, performance, and system building. Specifically, this year the topics of interest of OPODIS included:

- Design and analysis of distributed algorithms
- Synchronization, concurrent algorithms, shared and transactional memory
- Design and analysis of concurrent and distributed data structures
- Communication networks (protocols, architectures, services, applications)
- High-performance, cluster, cloud and grid computing
- Mesh and ad-hoc networks (wireless, mobile, sensor), location and context-aware systems
- Mobile agents, robots, and rendezvous
- Internet applications, social systems, peer-to-peer and overlay networks
- Distributed operating systems, middleware, and distributed database systems
- Programming languages, formal methods, specification and verification applied to distributed systems
- Embedded and energy-efficient distributed systems
- Distributed event processing
- Distributed storage and file systems, large-scale systems, and big data analytics
- Dependable distributed algorithms and systems
- Self-stabilization, self-organization, autonomy
- Security and privacy, cryptographic protocols
- Game-theory and economical aspects of distributed computing
- Randomization in distributed computing
- Biological distributed algorithms

We received 82 submissions, each of which was reviewed by at least four members of the Program Committee with the help of external reviewers. Overall, the quality of the submissions was very high. Out of the 82 submissions, 30 papers were selected to be included in these proceedings.

Following last year's practice, this edition of OPODIS proceedings appears in the Leibniz International Proceedings in Informatics (LIPIcs) series. LIPIcs proceedings are available online and free of charge to readers, and the production costs are paid in part from the conference budget. The review process was done using EasyChair.

The Best Paper Award was given to Keren Censor-Hillel and Michal Dory for the paper "Fast Distributed Approximation for TAP and 2-Edge-Connectivity".

This year OPODIS had two distinguished invited keynote speakers: Luís E. T. Rodrigues (INESC-ID, IST-UL, Portugal) and Roger Wattenhofer (ETHZ, Switzerland).

We would like to thank all authors for submitting their work to OPODIS. We are also grateful to the members of the Program Committee for their hard work in reviewing papers

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS 2017

and their active participation in the online discussions. We also thank the external reviewers for their help with the reviewing process.

Organizing this event would not have been possible without the time and the effort of the Organizing Committee, notably Ibéria Medeiros responsible for the local arrangements and web site, Miguel Matos who handled publicity, and João Leitão who managed the proceedings.

On behalf of all participants we thank them for their work that made the conference a truly enjoyable event beyond the scientific and technical aspects.

Finally, we would like to thank the Steering Committee members for their valuable advice and all the sponsors for their support.

December 2017

James Aspnes (Yale University)
Alysson Bessani (Faculdade de Ciências, Universidade de Lisboa)
Pascal Felber (University of Neuchâtel)

Program Committee

General Chair

Alysson Bessani, Faculdade de Ciências, Universidade de Lisboa, Portugal

Program Chairs

Pascal Felber, University of Neuchâtel, Switzerland

James Aspnes, Yale University, USA

Program Committee

Sara Bouchenak, INSA Lyon, France

Armando Castaneda, UNAM.Mexico

Bogdan Chlebus, University of Colorado, USA

Xavier Defago, Tokyo Institute of Technology, Japan

Carole Delporte, Université Paris Diderot, France

Oksana Denysyuk, Pure Storage, USA

David Doty, University of California, Davis

Jim Dowling, SICS/KTH, Swedish

Patrick Eugster, TU Darmstadt, Germany

Chryssis Georgiou, University of Cyprus, Cyprus

Wojciech Golab, University of Waterloo, Canada

Vincent Gramoli, University of Sydney, Australia

David Ilcinkas, CNRS, Bordeaux, France

Taisuke Izumi, Nagoya Institute of Technology, Japan

Ruediger Kapitza, TU Braunschweig, Germany

Christoph Lenzen, MPI for Informatics, Germany

Rui Oliveira, Universidade do Minho, Portugal

Emanuel Onica, Alexandru Ioan Cuza University of Iasi, Romania

Fernando Pedone, University of Lugano, Switzerland

Sebastiano Peluso, Virginia Tech, USA

Peter Pietzuch, Imperial College London, England

Maria Potop-Butucaru, Université Paris-VI Pierre-et-Marie-Curie, France

Laurent Réveillère, Bordeaux INP/Labri, France

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS 2017

Etienne Rivière, Université Catholique de Louvain, Belgium

Luís Rodrigues, INESC-ID / IST - University of Lisboa, Portugal

Paolo Romano, INESC-ID / IST - University of Lisboa, Portugal

Romain Rouvoy, University of Lille / Inria / IUF, France

Christian Scheideler, University of Paderborn, Germany

Paul Spirakis, University of Liverpool, UK and University of Patras, Greece

Jukka Suomela, Aalto University, Finland

Maarteen van Steen, University of Twente, Netherlands

Roman Vitenberg, University of Oslo, Norway

Spyros Voulgaris, VU University Amsterdam, Netherlands

Yukiko Yamauchi, Kyushu University, Japan

Haifeng Yu, National University of Singapore, Singapore

Steering Committee

Christian Cachin, IBM Research, Zurich, Switzerland

Panagiota Fatourou, FORTH ICS & University of Crete, Greece

Alessia Milani, University of Bordeaux, France

Fernando Pedone, University of Lugano, Switzerland

Maria Potop-Butucaru, Université Paris-VI Pierre-et-Marie-Curie, France

Giuseppe Prencipe, Università di Pisa , Italy

Etienne Rivière, Université Catholique de Louvain, Belgium

Sebastien Tixeuil, IUF & Université Pierre et Marie Curie - Paris 6, France (chair)

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS 2017

Organization Committee

Ibéria Medeiros (local arrangements chair), LASIGE & FCUL, University of Lisboa, Portugal

Miguel Matos (publicity chair), INESC-ID & IST, University of Lisboa, Portugal

João Leitão (publication chair), NOVA LINCS & FCT, NOVA University of Lisboa, Portugal

List of Authors

Ittai Abraham
VMware Research, Palo Alto, USA
iabraham@vmware.com

Eduardo Alchieri
University of Brasília, Brazil
alchieri@unb.br

Hagit Attiya
Department of Computer Science, Technion, Haifa 32000, Israel
hagit@cs.technion.ac.il

Alysson Bessani
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
anbessani@ciencias.ulisboa.pt

Roderick Bloem
TU Graz, Inffeldgasse 16a/II, 8010 Graz, Austria
roderick.bloem@iaik.tugraz.at

François Bonnet
Graduate School of Engineering, Osaka University, Osaka, Japan
francois@cy2sec.comm.eng.osaka-u.ac.jp

Conrad Burchert
ETH Zurich, Switzerland
bconrad@ethz.ch

Gonçalo Cabrita
NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal
g.cabrita@campus.fct.unl.pt

Keren Censor-Hillel
Technion, Department of Computer Science, Haifa, Israel
ckeren@cs.technion.ac.il

Himanshu Chauhan
University of Texas at Austin, USA
himanshu@utexas.edu

Bogdan S. Chlebus
Department of Computer Science and Engineering, University of Colorado Denver, Denver, Colorado 80217, USA
bogdan.chlebus@ucdenver.edu

Huda Chuangpishit
Department of Computer Science, Concordia University, Montreal, Canada
hoda.chuang@gmail.com

Nachshon Cohen
Computer Science Dept., EPFL
nachshonc@gmail.com

Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
miguel.p.correia@tecnico.ulisboa.pt

Gianluca De Marco
Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, 84084 Salerno, Italy
demarco@dia.unisa.it

Giuseppe A. Di Luna
University of Ottawa, Ottawa, Canada
gdiluna@uottawa.ca

Michael Dinitz
Johns Hopkins University, Baltimore, MD, USA
mdinitz@cs.jhu.edu

Ha Thi Thu Doan
Japan Advanced Institute of Science and Technology, Nomi, Japan
doanha@jaist.ac.jp

Stefan Dobrev
Slovak Academy of Sciences, Bratislava, Slovakia

Michal Dory
Technion, Department of Computer Science, Haifa, Israel
smichald@cs.technion.ac.il

Rotem Dvir
The Interdisciplinary Center P.O.Box 167, Herzliya 46150, Israel
rotem.dvir@gmail.com

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS 2017

Panagiota Fatourou
Institute of Computer Science (ICS), Foundation of Research and Technology-Hellas (FORTH), and Department of Computer Science, University of Crete, Greece
faturu@csd.uoc.gr

Paola Flocchini
University of Ottawa, Ottawa, Canada
paola.flocchini@uottawa.ca

Roy Friedman
Department of Computer Science, Technion
roy@cs.technion.ac.il

Arie Fouren
Faculty of Business Administration, Ono Academic College, Kiryat Ono, 5545173, Israel
aporan@ono.ac.il

Joni da Silva Fraga
Federal University of Santa Catarina, Brazil
fraga@das.ufsc.br

Vijay K. Garg
University of Texas at Austin, USA
garg@ece.utexas.edu

Tzlil Gonen
Tel Aviv University
tzlilgon@mail.tau.ac.il

Fabíola Greve
Federal University of Bahia, Brazil
fabiola@dcc.ufba.br

Maurice Herlihy
Computer Science Dept., Brown University
mph@cs.brown.edu

Aurélie Hurault
IRIT - Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France

Damien Imbs
LIF, Aix-Marseille Université & CNRS, France
damien.imbs@lif.univ-mrs.fr

Michiko Inoue
Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan
kounoe@is.naist.jp

André Joaquim
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
andre.joaquim@tecnico.ulisboa.pt

Tim Jungnickel
Technische Universität Berlin, Germany
tim.jungnickel@tu-berlin.de

Nikolaos D. Kallimanis
Institute of Computer Science, Foundation of Research and Technology-Hellas
nkallima@ics.forth.gr

Shuji Kijima
Kyushu University, Fukuoka, Japan
kijima@inf.kyushu-u.ac.jp

Igor Konnov
TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
konnov@forsyte.at

Janne H. Korhonen
Aalto University, Finland
janne.h.korhonen@aalto.fi

Rastislav Královič
Comenius University, Bratislava, Slovakia

Fabian Kuhn
University of Freiburg and Bremen University, Germany
kuhn@cs.uni-freiburg.de

Petr Kuznetsov
LTCI, Télécom ParisTech, Université Paris Saclay, France
petr.kuznetsov@telecom-paristech.fr

Marijana Lazić
TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
lazic@forsyte.at

Tuomo Lempiäinen
Department of Computer Science, Aalto University, Espoo, Finland
tuomo.lempiainen@aalto.fi

Roni Licher
Department of Computer Science, Technion
ronili@cs.technion.ac.il

Matthias Loibl
Technische Universität Berlin, Germany
matthias.loibl@mailbox.tu-berlin.de

Victor Luchangco
Oracle Labs, Burlington, USA
victor.luchangco@oracle.com

Dahlia Malkhi
VMware Research, Palo Alto, USA
dmalkhi@vmware.com

Saeed Mehrabi
School of Computer Science, Carleton University, Ottawa, Canada
saeed.mehrabi@carleton.ca

Lata Narayanan
Department of Computer Science, Concordia University, Montreal, Canada
lata@cs.concordia.ca

Kartik Nayak
University of Maryland, College Park, USA
kartik@cs.umd.edu

Yasamin Nazari
Johns Hopkins University, Baltimore, MD, USA
ynazari@jhu.edu

Susumu Nishimura
Dept. of Mathematics, Graduate School of Science, Kyoto University, Japan
susumu@math.kyoto-u.ac.jp

Kazuhiro Ogata
Japan Advanced Institute of Science and Technology, Nomi, Japan
ogata@jaist.ac.jp

Lennart Oldenburg
Technische Universität Berlin, Germany
l.oldenburg@mailbox.tu-berlin.de

Fukuhito Ooshita
Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan
f-oosita@is.naist.jp

Jaroslav Opatrny
Department of Computer Science, Concordia University, Montreal, Canada
opatrny@cs.concordia.ca

Rotem Oshman
Tel Aviv University
roshman@mail.tau.ac.il

Miguel L. Pardal
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
miguel.pardal@tecnico.ulisboa.pt

Dana Pardubská
Comenius University, Bratislava, Slovakia

Erez Petrank
Computer Science Dept., Technion
erez@cs.technion.ac.il

Nuno Preguiça
NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal
nuno.preguica@fct.unl.pt

Philippe Quéinnec
IRIT - Universiteé de Toulouse, 2 rue Camichel, F-31000 Toulouse, France

Ling Ren
Massachusetts Institute of Technology, Cambridge, USA
renling@mit.edu

Thibault Rieutord
LTCI, Télécom ParisTech, Université Paris Saclay, France
thibault.rieutord@telecom-paristech.fr

Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
ler@tecnico.ulisboa.pt

Joel Rybicki
University of Helsinki, Finland
joel.rybicki@helsinki.fi

Nicola Santoro
Carleton University, Ottawa, Canada
santoro@scs.carleton.ca

Philipp Schneider
University of Freiburg, Germany
philipp.schneider@cs.uni-freiburg.de

Adam Shimi
IRIT - Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France

Michael Spear
Lehigh University, Bethlehem, USA
spear@lehigh.edu

Alexander Spiegelman
Technion, Haifa, Israel
sasha.speigelman@gmail.edu

Jukka Suomela
Department of Computer Science, Aalto University, Espoo, Finland
jukka.suomela@aalto.fi

Muhammed Talo
Bilgisayar Muühendisliği, Munzur Üniversitesi, 62000 Tunceli, Turkey
muhammedtalo@munzur.edu.tr

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

Yusaku Tomita
Kyushu University, Fukuoka, Japan
tomita@tcslab.csce.kyushu-u.ac.jp

Giovanni Viglietta
University of Ottawa, Ottawa, Canada
gvigliet@uottawa.ca

Elias Wald
Computer Science Dept., Brown University
elias_wald@brown.edu

Roger Wattenhofer
ETH Zurich, Switzerland
wattenhofer@ethz.ch

Josef Widder
TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
widder@forsyte.at

Ken'ichi Yamaguchi
College of Information Science, National Institute of Technology, Nara College, Nara, Japan
yamaguti@info.nara-k.ac.jp

Masafumi Yamashita
Kyushu University, Fukuoka, Japan
mak@inf.kyushu-u.ac.jp

Yukiko Yamauchi
Kyushu University, Fukuoka, Japan
yamauchi@inf.kyushu-u.ac.jp

Hiroto Yasumi
College of Information Science, National Institute of Technology, Nara College, Nara, Japan
a0858@stdmail.nara-k.ac.jp

Gili Yavneh
Department of Computer Science, Technion
giliyav@cs.technion.ac.il

Tingzhe Zhou
Lehigh University, Bethlehem, USA
tiz214@lehigh.edu

Causality for the Masses: Offering Fresh Data, Low Latency, and High Throughput

Luís Rodrigues

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
ler@tecnico.ulisboa.pt

Abstract

The problem of ensuring consistency in applications that manage replicated data is one of the main challenges of distributed computing. Among the several invariants that may be enforced, ensuring that updates are applied and made visible respecting causality has emerged as a key ingredient among the many consistency criteria and client session guarantees that have been proposed and implemented in the last decade.

Techniques to keep track of causal dependencies, and to subsequently ensure that messages are delivered in causal order, have been widely studied. It is today well known that, in order to accurately capture causality one may need to keep a large amounts of metadata, for instance, one vector clock for each data object. This metadata needs to be updated and piggybacked on update messages, such that updates that are received from remote datacenters can be applied locally without violating causality. This metadata can be compressed; ultimately, it is possible to preserve causal order using a single scalar as metadata, i.e., a Lamport's clock. Unfortunately, when compressing metadada it may become impossible to distinguish if two events are concurrent or causally related. We denote such scenario a *false dependency*. False dependencies introduce unnecessary delays and impair the latency of update propagation. This problem is exacerbated when one wants to support partial replication.

Therefore, when building a geo-replicated large-scale system one is faced with a dilemma: one can use techniques that maintain few metadata and that fail to capture causality accurately, or one can use techniques that require large metadata (to be kept and exchanged) but have precise information about which updates are concurrent. The former usually offer good throughput at the cost of latency, while the latter offer lower latencies sacrificing throughput. This talk reports on Saturn[1] and Eunomia[2], two complementary systems that break this tradeoff by providing simultaneously high-throughput and low latency, even in face of partial replication. The key ingredient to the success of our approach is to decouple the metadata path from the data path and to serialize concurrent events (to reduce metadata), in the metadata path, in a way that minimizes the impact on the latency perceived by clients.

1998 ACM Subject Classification C.2.4 Distributed Systems, H.2.4 Systems

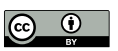
Keywords and phrases Distributed Systems, Causal Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.1

References

1 Manuel Bravo, Luís Rodrigues, and Peter van Roy. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the EuroSys 2017*, Belgrade, Serbia, 2017.

2 Chathuri Gunawardhana, Manuel Bravo, and Luís Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara (CA), USA, 2017.



piChain: When a Blockchain Meets Paxos

Conrad Burchert[1] **and Roger Wattenhofer**[2]

1 **ETH Zurich, Switzerland**
bconrad@ethz.ch
2 **ETH Zurich, Switzerland**
wattenhofer@ethz.ch

Abstract

We present a new fault-tolerant distributed state machine to inherit the best features of its "parents in spirit": Paxos, providing strong consistency, and a blockchain, providing simplicity and availability. Our proposal is simple as it does not include any heavy weight distributed failure handling protocols such as leader election. In addition, our proposal has a few other valuable features, e.g., it is responsive, it scales well, and it does not send any overhead messages.

1998 ACM Subject Classification C.4 Performance of Systems

Keywords and phrases Consensus, Crash Failures, Availability, Network Partition, Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.2

1 Introduction

Two fundamental philosophies to build fault-tolerant distributed state machines exist. One is mentioned in every other newspaper: The *blockchain* [22] is a fault-tolerant data structure to organize transactions. Its main advantage is its simplicity; the main disadvantage is that a blockchain is only eventually consistent.

On the other hand, we have the various consensus and agreement protocols designed by the distributed systems community, e.g., *Paxos* [16]. These protocols usually provide strong consistency, but have scalability issues. The idea of this paper is to marry the two worlds in a natural way, inheriting the best features of both.

We present *piChain*, a fault-tolerant distributed state machine (also known as repeated consensus, agreement, ledger, log, history, event sourcing) based on a blockchain, with integrated strong consistency. Our proposal is:

- Fault-Tolerant: piChain can handle various types of faults, e.g., crashes, crash-recoveries, message omissions, network partitions. It *cannot* handle arbitrarily malicious ("byzantine") faults, as we believe that there is a performance penalty many applications are not willing to pay.
- Fast: The basic functionality has no overhead; transactions can be created as fast as they can be sent and received by the network. Strong consistency will usually be achieved in one message round-trip time.
- Quiet: If no new transactions are created, no messages need to be sent. In other words, piChain needs no heartbeat.
- Scalable: piChain works with just a few nodes as well as hundreds of nodes, in a single location as well as distributed around the globe. There are no subroutines that produce a quadratic number of messages. If shooting for scalability, each node needs to send and receive only a few messages per transaction.

- Light: In comparison to other protocols, piChain is a light protocol, e.g., it does not have an explicit leader election subroutine. As such, the piChain architecture should be simple to understand and modify.
- Available and Consistent: piChain provides *both* availability and strong consistency. As such it will continue to try to process transactions even if the network is partitioned, and only short intervals of connectivity of a majority of nodes are enough to commit again to a globally consistent state. As such piChain works in harsh networking environments.

There are many applications that may benefit to have both availability and strong consistency. For example, take an airline reservation system with two data centers. For each flight, each data center may initially have an allowance of half the available seats. Even if the two data centers are temporarily not connected because of a network partition, both data centers can decide to sell seats based on their local allowance. When the whole system is reconnected, the data centers can regain strong consistency, e.g., re-balance their allowance.

In the next section we present the architecture of piChain. Section 3 explains how piChain differs from previous protocols. In Section 4 we explain why piChain is correct and efficient, and in Section 5 we evaluate our implementation.

2 Architecture

In this section we describe the three ingredients of the piChain architecture, and how they interact with each other.

2.1 Transactions and Blocks

We want to order and store *transactions*. Transactions can be, e.g., executable commands in a storage system, or financial transactions. Apart from its content, each transaction includes a unique ID, which is a combination of the unique ID of the node that created the transaction, and a sequence number.[1] Transactions are being sent to all the nodes in the system.[2]

Transactions are grouped in *blocks*, a block may contain many or just a single transaction. Blocks are created by arbitrary nodes, like transactions they contain an ID, which is again a combination of the ID of the creator of the block and a sequence number.

In addition, each block contains a pointer to a parent block.[3] The parent of a newly created block is the deepest block the creator has seen. What is the *deepest* block? Each block contains a depth field, which is the total number of transactions the block and all its ancestor blocks contain.[4] If two blocks have exactly the same depth, we will use the unique block ID to break ties.

---

[1] The sequence number is simply how many transactions that node already created. These sequence numbers help nodes to recover missing messages, e.g. if a node v has seen a transaction with ID (u,7) by node u but not transaction (u,6), node v can ask node u or any other node about the missing transaction.

[2] If the system consists of just a few nodes, all messages are sent directly. If the system consists of hundreds of nodes, one should rather use an overlay, and send all messages between neighbors using flooding. Thanks to flooding, each node must only transmit or receive a constant number of messages per transaction.

[3] We use the usual family relations to describe relations between blocks, in particular parent, ancestor, and descendent.

[4] In other words, the depth of a block b is the sum of the depth of b's parent and the number of transactions block b contains.

The root block is already available when the system is initialized; the root block has no transactions, no parent, and its depth is 0. The transactions of a block may not contradict the transactions of the blocks on the path to the root.[5] After its creation a block is sent to all nodes in the system.

In piChain, any node can create a block, the only question is *when*.[6]

2.2 Node States

Each node is in one of three states: quick, medium, or slow. This state describes the node's eagerness to create a block.

If a node is *quick*, it will create blocks quickly: whenever a quick node sees a new transaction that it has not yet seen included in a block, it will instantly create a block with this transaction.[7]

A *slow* node will only create a block if it has waited for a considerable time after seeing a new transaction that is not yet in a block. The earliest time t for a slow node to consider creating a block is when it should have received the block already by either a quick or a medium node. In addition to this earliest time t, slow nodes will also randomly wait even longer, long enough that only one (the fastest) slow node s will create a block in expectation, as other slow nodes will see that block by s before they will create a block.

In contrast to quick and slow nodes, *medium* nodes only exist transiently. After learning about a new transaction, a medium node waits until it should have received a block with this transaction by a quick node.

Each node will upgrade its state as follows:

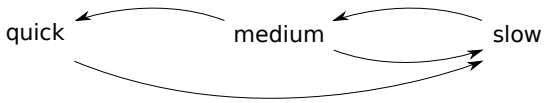


Figure 1 The state transitions: A node promotes itself to the next faster state when it creates a block. A node demotes itself to slow if it sees a block b by some other node, and either the creator of b is quick or b is the new deepest block.

2.3 Strong Consistency

Whenever a quick node creates a block and is not already in the process of committing another block, it may decide to commit this block. Committing a block commits all the transactions in that block, and all the transactions in all the blocks on the path from the root to that block.[8] Committed blocks (transactions) are final and cannot be uncommitted again. A committable block must be a descendant of all previously committed blocks, i.e., the committed blocks are totally ordered in tree of blocks. The first committed block is the root block; the root block is the *precursor* of the second committed block, which in turn is

[5] E.g., a transaction that commands to move a file cannot be included in a block if its parent block included a transaction that deleted the file.

[6] In contrast to the Bitcoin blockchain, nodes do not have to perform a proof of work in order to generate a new block.

[7] A quick node may also wait a bit to accumulate several transactions; however, other nodes must know about such an intentional offset and adapt their timings accordingly.

[8] Many of which may already be committed.

Algorithm 1 Committing: Paxos with Blocks.

Quick Node	**All Nodes**
b_{prop} = deepest block	$b_{\max} = \perp$
$b_{\text{com}} = \perp$	$b_{\text{prop}} = \perp$
	$b_{\text{supp}} = \perp$
Phase 1	
1: Send try(b_{new}) to all nodes	
	2: On receiving a try(b_{new}) message:
	3: **if** b_{new} deeper than $b_{\max}$ **then**
	4: $b_{\max} = b_{\text{new}}$
	5: Answer with ok(b_{prop}, b_{supp})
	6: **end if**
Phase 2	
7: Majority responded with ok(b_{prop}, b_{supp}):	
8: $b_{\text{com}} = b_{\text{new}}$	
9: **if** some response included $b_{\text{prop}} \neq \perp$ **then**	
10: $b_{\text{com}} = b_{\text{prop}}$ with deepest b_{supp}	
11: **end if**	
12: Send propose(b_{com}, b_{new}) to all nodes	13: On receiving a propose(b_{com}, b_{new}) message:
	14: **if** $b_{\text{new}} = b_{\max}$ **then**
	15: $b_{\text{prop}} = b_{\text{com}}$
	16: $b_{\text{supp}} = b_{\text{new}}$
	17: Answer with ack(b_{com})
	18: **end if**
Phase 3	
19: Majority responded with ack(b_{com}):	
20: Send commit(b_{com}) to all nodes	

again the precursor of the third committed block. In other words, every committed block except the root block have the previous committed block as their precursor.

In order to commit a block, a quick node must convince a majority of nodes twice. This works trivially if there is no competition, i.e., if there is a single quick node. If no node is quick, blocks will transiently not be committed, even though new blocks are still created. If there are multiple quick nodes (e.g. after a network partition), our protocol still works, as it is a variant of Paxos, formally described in Algorithm 1.

In Algorithm 1, each node stores a list of already committed blocks, initially only the root block is committed. For *every* committed block b, each node stores three variables $b_{\max}$ (the deepest block seen in round 1), b_{prop} (a proposed block) and b_{supp} (a block supporting the proposed block). These are initially $\perp$ for every block, including the root. In addition, a

quick node q that initiates a the commit protocol temporarily also stores b_{com} (a compromise block).

Every committed block (but the root block) has a precursor block. In order to commit a new block b_{new}, a quick node needs to refer the last already committed block b, the precursor block of the block to be committed. In order to avoid notational clutter in Algorithm 1, we omitted all the precursor information. When we write that a node sends b_{x}, we mean that the node sends both the ID of its precursor block b (for reference) and the value of b_{x}.

Moreover, Algorithm 1 can be pipelined: A quick node that passed phase two can already start committing the next block. This becomes even more powerful with an implicit phase 1. Every propose for a block can implicitly be a try($\perp$) with an empty block of depth 0. This way a successful phase 2 for a block always includes a successful phase 1 for the next round and the quick node can directly propose a successor block again. In the regular case the quick node is only sending one message type, a pipelined/truncated combination of a phase 3 (line 20) and phase 2 (line 12) message: "commit(b) and propose(b_{new}, $\perp$)". If some other node intervenes with its own try message and reaches a majority in the first phase, the quick node's propose will fail in line 14 and the quick node has do to an explicit new phase 1.

After a block b is committed, there might be blocks which are neither a descendant of b nor on the path from the root block to b. These blocks are removed; the transactions inside these blocks may however be salvaged (if they do not contradict the transactions of the committed blocks), by simply creating a new block with these transactions as a descendent of b. Committing a block is also an opportunity to compact the log up to block b.

Finally, we could also use commits to implement node membership changes; we simply add a new node u with a transaction t, and node u will be included as a voting node in Algorithm 1 as soon as transaction t has been committed.

3 Related Work

There is no lack of protocols that try to solve the same problem as we do, e.g., Chubby [4], Zookeeper [14], Spanner [7], CORFU [2]. The most recent heavy weight champion of this class is probably Raft [23]. Indeed, Raft has been designed with a similar agenda in mind (simplicity first, strong consistency, explicit timing, no byzantine failures). The main argument in favor of piChain's simplicity is its "lightness": Raft (and its competitors) uses leaders, and these leaders have their epochs (or terms) when they are ruling. Whenever a leader is not responsive anymore, other nodes have to notice this first, then they have to agree that they want a new leader, at which point a leader election algorithm is started.[9] In piChain, each node just decides by itself that something needs to be done: it directly promotes or demotes itself without communication. Moreover, piChain is quiet in the sense that it does not send any messages if there are no new transactions, whereas Raft needs some kind of heartbeat. Finally, piChain provides scalability and availability out of the box.

The only heavy part of piChain is Algorithm 1, a blockchain version of Paxos [12, 16]. In our opinion, Paxos and blockchains are a natural fit, since a blockchain orders transactions which is then augmented to strong consistency with Paxos. In contrast to Paxos, piChain introduces a new way to prevent proposers from interrupting ongoing commits. Lamport's Paxos cared about *correctness* rather than efficiency. Paxos is a truly seminal protocol that is hard to get around these days.[10] One may claim that Lamport's original publication [16]

[9] We would argue that the leader election algorithm of Raft is similar to the first Paxos round.
[10] When working on piChain we tried to deviate from Paxos more radically, without success.

left message timing as a exercise to the reader, and that piChain is just an explanation how to implement a repeated version of Paxos also known as multi-Paxos. Paxos was studied in various dimensions over the years, e.g., [6, 17, 19, 21, 25]. We do not use any of these improvements, just the original.

Depending on the application, commits do not have to be done often but just from time to time, to shorten the blockchain. If the application depends on fast commits, we can also commit each and every block. As we will discuss in Section 4, thanks to majority-only voting, pipelining and truncating, piChain will be faster than three-phase commit [26].

In addition, piChain cares about availability, something Paxos did not bother with. After Brewer [3] explained that consistency is not everything and availability should not be forgotten, the pendulum is swinging back and forth between strong consistency and high availability. We believe that there is value in having both, as many applications may want to implement decisions even if they are not final.

Speaking of high availability: "Satoshi Nakamoto" introduced the concept of a blockchain in a byzantine setting [22]. While his proof-of-work based system can tolerate byzantine failures with anonymous nodes, generating blocks cannot be fast because of forks, and blocks are never really committed [8, 10, 13].

Previous work tried to prevent forks of the Bitcoin blockchain by using strongly consistent byzantine agreement to agree on blocks before adding them to the blockchain [9]. This is implemented in various cryptocurrencies, e.g., the masternode system of Dash [11]. While the byzantine setting is more difficult than piChain, those systems cannot recover from a state of too many crashed nodes, as the block creators are themselves elected in blocks. The system uses strong consistency to append to the blockchain. We believe that piChain is more natural, ordering by the blockchain first and strong consistency later.

In the last 20 years we have seen an army of new protocols that try to cope with byzantine (arbitrary, malicious) failures, e.g., PBFT [5], Farsite [1], Zyzzyva [15] and a byzantine version of Paxos [18]. These protocols build on the earlier theoretical work, among others again by Lamport [20, 24]. In many applications one must be able to tolerate byzantine behaviour. However, Google or Amazon have developed their fault-tolerant distributed systems to handle crash failures only. We believe that the cost of fully byzantine protocols may be prohibitive for applications which need to be efficient. Nevertheless, one can add byzantine tolerant elements to piChain, e.g. when creating a new transaction, nodes could be asked to cryptographically authenticate the transaction.

4 Analysis

In this section we discuss why piChain is correct and efficient.

4.1 Transactions and Blocks

Let us start with some basics about transactions and blocks. First note that the blocks form a tree: Every block has a parent, which is another block that has been created earlier. By induction following the parent pointers brings us to earlier and earlier blocks, and ultimately to the root block.

Also, every transaction will be in a block: Even if it is a slow node, the creator of the transaction will include it in a block after it has not seen it in a block for long enough. If a transaction is in a block b that is discarded because of committing a block which is neither an ancestor nor a descendent of b, then the transaction is again on the market. If the transaction does not contradict an already committed transaction, the nodes will put it in a block again.

4.2 Node States

The waiting times of quick, medium, and slow nodes until they create a block are crucial to the performance of piChain:

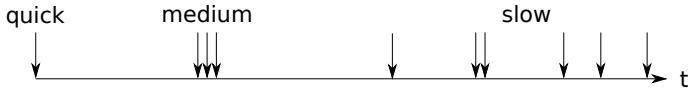


Figure 2 Waiting time of nodes in different states: A quick node creates a block instantly when seeing a new transaction. A medium node waits long enough to ensure not to compete with a quick node. A slow node waits long enough to ensure to compete neither with a quick nor a medium node. The figure represents a situation with one quick, three medium, and some slow nodes.

The waiting times depend on the network characteristics. To understand this aspect better, let us first discuss a simplistic network model by assuming that the time to deliver a message between any two nodes is always exactly 1 time unit, in both directions. Quick nodes always immediately create a block when learning about a new transaction, also in this simplified model. After learning about a new transaction, a medium node u should wait time $1 + \epsilon$, unless the transaction was created by u, in which case u should wait time $2 + \epsilon$ (as this is enough time to send the transaction to the quick node and send a block with the transaction back). A slow node should wait time $3 + 2\epsilon + r$ (or $4 + 2\epsilon + r$ for a self-created transaction), where the parameter r is a random time chosen uniformly in the interval $[0, n + 1]$, where n is the number of nodes. In this simplified model, these timings are the shortest possible timings allowed by the description in Section 2.2, as they allow the faster nodes to deliver a block before a slower node will create one.

For example, let us assume that slow node s produces a transaction at time t, sending it to the other nodes. If there is a medium node m, node m will learn about this transaction at time $t + 1$, create a block at time $t + 3 + \epsilon$, which will arrive back at slow node s at time $t + 4 + \epsilon$, right before the earliest possible time s may create a block. Arguing all other cases is similar, we know that a quick node will not be challenged by a medium node, and a medium node will not be challenged by a slow node.

In absence of quick and medium nodes, the random time r for the slow nodes makes sure that there is a good chance only a single slow node creates a block. Moreover, using a standard Chernoff bound one can show that with high probability at most $O(\log n)$ slow nodes create a block within one time unit.

If there is exactly one quick node, and all others are slow, we call the system *healthy*.[11] Our system should almost always be healthy. The healthy state is stable, since the timings guarantee that the quick node will not be challenged by the other nodes. But what if something does not work as anticipated, e.g. the single quick node crashes?

If the single quick node crashes (or the network has problems and does not deliver the messages of the quick node), we have only slow nodes. If so, a slow node will jump in and produce a block. There might be several such slow nodes, but there will be few. If there is

[11] Because of this, one might consider naming quick nodes "leaders", slow nodes "followers", and medium nodes "candidates". However, the term leader usually implies that the system *guarantees* to have at most one. Our piChain architecture does not attempt to have such a guarantee, it does not even have an explicit leader election subroutine. Instead nodes just update their state (quick, medium, slow) locally without interacting with other nodes, so the classic terms seem inaccurate.

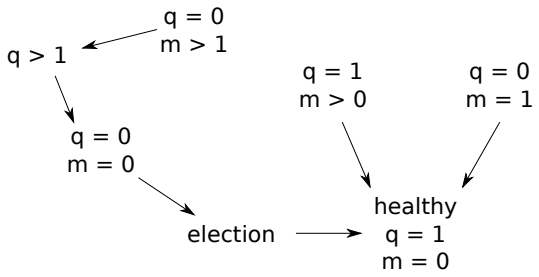


Figure 3 Recovery of the network to the healthy state. The nodes of this graph represent all possible states of the whole system: how many quick (q) nodes do we have and if relevant how many medium (m) nodes. The election state is special, as it is characterized by possibly multiple medium nodes, which have in-flight blocks. The in-flight blocks will demote all medium nodes except for the one that created the deepest block, so only that node will become quick.

more than one slow node challenging the crashed quick node, they will all send out their blocks more or less concurrently (otherwise they would had seen the other blocks before creating one themselves). According to Figure 1, all of these fastest slow nodes become medium nodes, but (unless there are further crashes or message omissions) all see all their blocks before they produce a next block. Only one of these blocks is the deepest (even if several have the same number of transactions we know that ties are broken by block ID), so only one now-medium node will create a second block and become quick. All other medium nodes see a deeper block and go back to slow again. This summarizes our implicit *election*.[12] This is probably the most regular way to get back to the healthy state, but others exist, as summarized in Figure 3.

So what if message delays are more realistic? The idea is to use previous measurements, like in TCP. If a non-quick node v learns about a transaction created by node u, node v considers the current message delays in the system. As described in Section 2.2, node v gives the faster nodes enough time to deliver their blocks, based on previous delivery times. If a node does not know anything about these times (e.g., the system just started), it can assume a very crude upper bound on message delivery time. We would suggest for a slow node v to wait for $2R + r \cdot 0.5R + 2\epsilon$, where r is again a random value in $[0, n + 1]$, and R is the absolute worst round-trip time any node has seen (or can imagine) when the network was not faulty. This way, we will be back in the healthy state in expected time $3.5R$ (we have $2.5R$ time until the first slow node s creates a block,[13], plus R time until node s creates a second block).

We could optimize R a bit more aggressively if we have previous timing measurements. Premature block creations will be handled gracefully in piChain, so they are no big deal really. On the other hand, these slow waiting times will happen so rarely that they barely matter for our overall system performance, so just choosing a high R is also fine. After all, the system will be mostly in the healthy state, where the single quick node does not wait at all.

[12] One might consider to "vaccinate" the healthy state: When we are in a healthy state, the single quick node q may consider choosing another node as its "heir"; this heir would then assume a medium state, and as such automatically be the first to create a block after the quick node crashed.

[13] If n nodes choose a random value in $[0, n + 1]$, the expected lowest value is 1.

4.3 Strong Consistency

This brings us to the third part of the architecture, the commits. First of all, if the system is healthy, Algorithm 1 reduces to a simple message ping pong: try-ok-propose-ack-commit not unlike three-phase-commit (3PC). One may argue that Algorithm 1 is faster than 3PC because the quick node does not have to wait for replies from *all* nodes, but merely a majority. Also, Algorithm 1 can be pipelined, and a stable quick node can directly enter round 2, i.e. in a healthy system, Algorithm 1 is truncated to lines 12–22.

If there are no quick nodes, we do not even attempt to commit a block. So the only interesting remaining cases are multiple competing quick nodes (after a partition), or if there are so many errors (crashes, message omissions) in the system that the quick node cannot easily finish its commit because it does not get the needed majorities.

The principle why the algorithm works is built around the implication of accepting a block proposal in round 2. A node u which answers ok in round 2 may have just been part of a successful commit.[14] To make sure only one block can be committed as a successor of a previous commit, node u must not accept a proposal for any other block until the node can be convinced that the commitment was unsuccessful. Luckily, any arriving propose message that proposes a different block with a deeper support proves that the last proposal was unsuccessful and the node can therefore safely support this new proposal. So in order to get a successful commit, we need a uninterrupted "double whammy" of rounds 1 and 2.

We can formally proof this intuition. Let $p = (b_{\text{com}}, b_{\text{new}})$ be a successful successor for committed block b, i.e., p was accepted by a majority of the nodes in round 2. Let p' be the first subsequent proposal. In order to not be ignored in round 1, p''s support needs to be deeper than p. Both p and p' are seen by a majority of nodes, so there is a node which has seen both p and p'. This node will report p to the quick node at the end of round 1. Therefore the quick node leading the proposal had both to choose from in line 10. If p' was for a block different than p that block must have been proposed with a deeper support block, as otherwise the quick node leading the commitment process would not have chosen it as the compromise block. However this contradicts the assumption that p' was the first proposal with a deeper support after p, therefore there can only be one successful proposal and thus a node which receives a new proposal with deeper support can assume all previous proposals failed.

This proof also applies to the truncated version, as the order of message arrivals is exactly the same. Every node simulates the arrival of a try message for the next round after every propose message is accepted, this way the quick node does not violate the protocol by omitting the first phase.

5 Evaluation

To test our implementation, we run it in a discrete event simulation to get accurate timings not affected by distributed time measuring difficulties. Most failures have no effect on the system, so we only tested a few rare but harsh scenarios. We have $n = 20$ nodes embedded randomly in a square with diagonal 0.5, with distance being message delay, so the maximum possible message delay is 0.5s. Transactions are created by random nodes at random times with an average rate of 10 transactions per second. The nodes use a worst-case round-trip time of $R = 1$s.

[14] Node u may not know that, though.

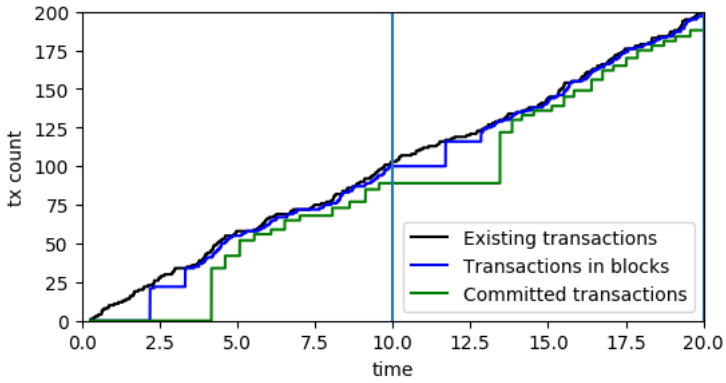


Figure 4 piChain with $n = 20$. After 10s the single quick node crashes. No new blocks are created until a slow node creates a block and becomes medium. A bit later, the same node becomes quick, and blocks are again created and committed with a fast pace.

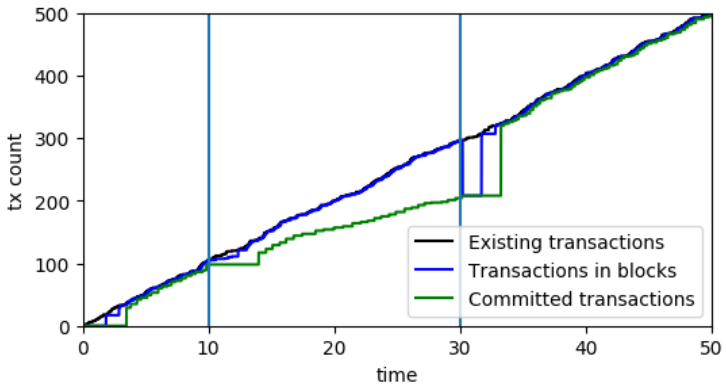


Figure 5 The network is partitioned into 8 resp. 12 nodes after 10s. Each side of the partition quickly finds a single quick node that is adding blocks, but only the majority side can commit its blocks. When the partition is resolved after 30s, the minority side temporarily loses its blocks because of the committed blocks on the majority side (see discussion in second paragraph of Section 4.1).

Figure 4 shows an example how the network recovers to healthy after the quick node crashed. Averaged over 100 runs of such crashes, the time until we are back to the healthy state is 3.67s on average, with a standard deviation of 0.49s. This conforms with our expectation of $3.5R$. The variations can be explained by how we measure. In particular, to get the protocol going, we need a transaction, and transactions first have to be created and delivered. Further randomness is introduced by different waiting times of slow nodes. Figure 5 presents the effects of a network partition.

In Section 4.2 we claimed that we always get back to the healthy state quickly. Figure 6 shows the actual times of the state chart of Figure 3.

In another experiment we analyze the behaviour of the algorithm in a highly unstable situation, where nodes repeatedly crash and recover; while being crashed all messages of a node are dropped. Two of the resulting plots are shown in Figure 7.

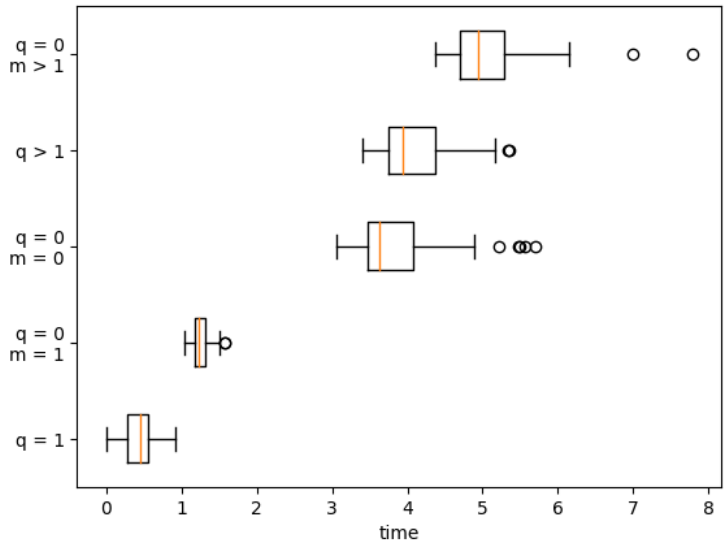


Figure 6 The time piChain needs to become healthy when being started in a random state. The start states are clustered according to Figure 3, e.g., when a third of the nodes is initially in each category quick, medium, and slow, we add a measurement to class $q > 1$.

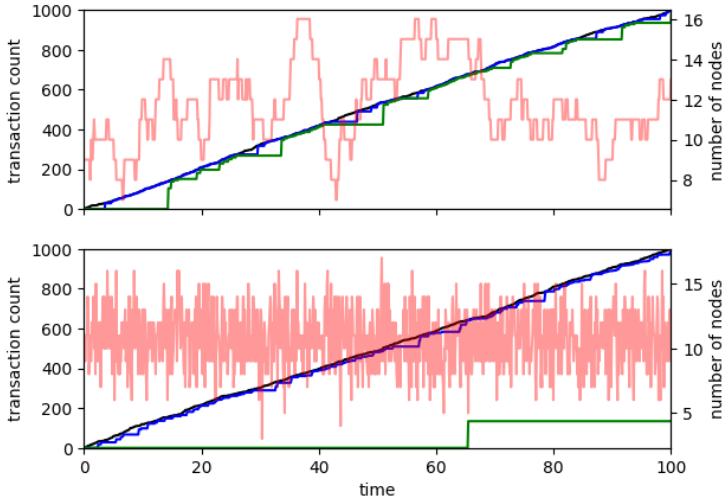


Figure 7 piChain with unstable nodes that crash and recover. The red curves show the number of currently working nodes. In the upper plot crashes last for an average of 20s, in the lower plot for 0.2s, but both plots have the same average number of 11 working nodes. One can see that blocks are generated quickly in both plots, but committing takes more time in the lower plot, since Algorithm 1 is often interrupted by crashes.

References

1 Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review (OSR)*, 2002.

2 Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei,

and John D. Davis. CORFU: A shared log design for flash clusters. In *Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

3 Eric A Brewer. Towards robust distributed systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

4 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating systems design and implementation (OSDI)*, 2006.

5 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

6 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

7 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database, 2012.

8 Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In *3rd Workshop on Bitcoin Research (BITCOIN)*, 2016.

9 Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2016.

10 Christian Decker and Roger Wattenhofer. Information Propagation in the Bitcoin Network. In *13th IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

11 Evan Duffield and Daniel Diaz. Dash: A privacy-centric crypto-currency, 2014.

12 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

13 Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

14 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference (USENIX ATC)*, 2010.

15 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review (OSR)*, 2007.

16 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

17 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

18 Leslie Lamport. Byzantizing paxos by refinement. In *International Symposium on Distributed Computing (DISC)*, 2011.

19 Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

20 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1982.

21 Iulian Moraru, David G Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *ACM Symposium on Cloud Computing*, 2014.

22 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

23 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

24 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 1980.

25 Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. In *International Conference on Very Large Data Bases (VLDB)*, 2011.

26 Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, 1983.

Broadcasting in an Unreliable SINR Model

Fabian Kuhn[1] and Philipp Schneider[2]

1 **University of Freiburg, Germany**
kuhn@cs.uni-freiburg.de
2 **University of Freiburg, Germany**
philipp.schneider@cs.uni-freiburg.de

Abstract

We investigate distributed algorithms for broadcasting in unreliable wireless networks. Our basic setting is the *signal to noise and interference ratio* (SINR) model, which captures the physical key characteristics of wireless communication. We consider a dynamic variant of this model in which an adversary can adaptively control the model parameters for each individual transmission. Moreover, we assume that the network devices have no information about the geometry or the topology of the network and do neither know the exact model parameters nor do they have any control over them.

Our model is intended to capture the inherently unstable and unreliable nature of real wireless transmission, where signal quality and reception depends on many different aspects that are often hard to measure or predict. We show that with moderate adaptations, the broadcast algorithm of Daum et al. [DISC 13] also works in such an adversarial, much more dynamic setting. The algorithm allows to broadcast a single message in a network of size n in time $\mathcal{O}(D \cdot \text{polylog}(n+R))$, where D is the diameter and R describes the granularity of the communication graph.

1998 ACM Subject Classification G.2.2 Network problems, F.2.2 Analysis of Algorithms

Keywords and phrases radio networks, wireless networks, broadcast, SINR model, unreliable communication, dynamic networks

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.3

1 Introduction

In the *signal to noise and interference* (SINR) model (a.k.a. the *physical interference model*), a message is received if and only if the ratio between the signal strength at the receiving node and the combined strength of the background noise and any interfering signals is above a given threshold. By now, the SINR model has become the standard communication model to study wireless network algorithms. In the distributed algorithms literature, different variants of the basic SINR model have been studied, based on the properties of the underlying geometric space, how much geometric information the network devices have and how much they know about the model parameters or the network topology.

One of the most general variants of the SINR model has been termed the *ad hoc* SINR model by Daum et al. in [4], where they study the problem of broadcasting a message to all nodes of a wireless network. In [4], it is assumed that the network nodes have no information about the geometry or the topology of the network, prohibiting algorithms that utilize advance knowledge about network topology and layout or the way that signals propagate in space. The distances between the nodes are assumed to form a general *growth-bounded* metric space. Furthermore, the nodes use uniform transmission powers, they have only approximate

knowledge of the hardware and model parameters, and they do not have additional capabilities like carrier sensing. Because the model makes only minimal assumptions, it allows to develop algorithms that can operate in a quite general setting.

However, while the authors of [4] assume that the model parameters are only known approximately, these parameters are still assumed to be uniform over the whole network and static over time. As a result, whether a message is successfully received is determined by a *deterministic* function depending on the system parameters and the geometry of the network. However in real wireless networks, communication often turns out to be rather unreliable and highly volatile, and system parameters tend to not be uniform over space and time [22]. As a result, practical wireless communication behaves in an inherently non-deterministic way. Examples for such unpredictable communication behavior are background noise due to coexisting networks or jamming, various multi-path effects due to changes of the environment, or fluctuations in sending power or signal sensitivity among different wireless devices.

In order to improve practical applicability in the present paper, we consider an unreliable variant of the ad hoc SINR model (which we will term the *unreliable SINR model*) by adding a worst-case adversary, which decides to a given extent whether or not messages are received. The adversary adds a dynamic and non-deterministic component to the model that allows to capture non-uniform and dynamically changing SINR parameters. More concretely, we assume that at each point in time and individually for each node, an adversary can adaptively control the threshold on the signal to interference and noise ratio above which a message can successfully be decoded (for a formal definition of the model, we refer to Section 3).

As our main contribution, we adapt the broadcast algorithm of Daum et el. [4] to work in the unreliable SINR model. We show that with relatively modest adaptions, the algorithm of [4] can also be used to efficiently solve the broadcast problem in the unreliable SINR model. More specifically, let G_C be the communication graph in which two nodes are adjacent if they can reliably communicate with each other in the unreliable SINR model (if the signal to noise ratio for the two nodes is sufficiently above the maximum threshold that the adversary is allowed to choose). We prove that the global broadcast problem can be solved in time $\mathcal{O}\big(D \cdot \text{polylog}(n+R) \cdot \frac{\beta_{\max}}{\beta_{\min}}\big)$, where D is the diameter of the communication graph G_C, R is the ratio of maximum to minimum distance between any two adjacent nodes in G_C,[1] and $[\beta_{\min}, \beta_{\max}]$ is the range within which the adversary can adaptively choose the SINR threshold in each time slot and for each node.

The paper is structured as follows. In Section 2 we summarize existing work related to broadcasting in the SINR model and non-deterministic distributed algorithms. In Section 3 we specify our model and give some notions used throughout the following sections. Section 4 gives an overview on the neighborhood dissemination process, which conducts one hop of the global message broadcast. In Section 5 to Section 8 we analyze the neighborhood dissemination protocol. Finally, in Section 9, we show that our abstract model, where the adversary can only control the SINR threshold, in fact allows to also capture a more general adversary, which controls all SINR-parameters.

[1] Theoretically, any functional dependency $R = f(n)$ is possible, e.g., for a series of nodes $v_1, \ldots, v_{n+1}$ with decreasing distances $d(v_i, v_{i+1}) = 1/f(i)$. However, we consider $R \in \omega(n)$ unlikely in practice. E.g., placing n nodes uniformly at random inside a square with sidelength $r_e/\sqrt{2}$ (where r_e is the effective communication range), yields an expected granularity $R \in \Theta(n)$.

2 Related Work

Broadcasting in the SINR Model. Broadcasting algorithms in conjunction with the SINR model were first investigated in [10]. They consider local broadcast, where each node needs to broadcast a message to all neighbors in the communication graph. Note that local broadcast can be used as a building block to solve global broadcast. Subsequent publications on broadcasting (local and global) in the SINR model include [13, 14, 25, 26]. However, all the aforementioned publications leverage assumptions that are incompatible with our conception of the SINR model. Incongruities include in particular that nodes have knowledge of their position or of distances to other nodes or that nodes can modify their sending power or have carrier sensing capability.

Broadcasting in the Ad Hoc SINR Model. Most relevant for our work are distributed information dissemination algorithms that work in the ad hoc SINR model as discussed in Section 1. The first such algorithm is the local broadcast algorithm of [12], which can be used to solve global broadcast in the ad hoc SINR model in time $\mathcal{O}(D\Delta_C(\log n)^2)$, where Δ_C is the maximum degree of the communication graph G_C. Note that Δ_C is a potentially large factor (e.g., in single-hop networks). In [4] the global broadcast problem is solved directly in time $\mathcal{O}(D \cdot \text{polylog}(n+R))$. Recall that R is the ratio between the largest and smallest distance between neighbors in the communication graph. The solution of [4] forms the basis of our algorithm and we will point out the differences to the algorithm and analysis alongside.

For large and moderately large values of R, the result of [4] has been improved by Jurdzinski, Kowalski, Rozanski, and Stachowiak [15], where it is shown that one can get rid of the dependency on R and solve the global broadcast problem in time $\mathcal{O}(D\log^2 n)$.[2] The algorithm of [15] is based on finding an assignment of probabilities such that in each local neighborhood, the sum of probabilities is upper and lower bounded by some constant.

Halldorsson et al. [11] use the concept of *abstract Medium Access Control* (abstract MAC) layers introduced in [16] (and subsequently enhanced in [7]), to implement higher-level procedures such as single and multi-message broadcast and consensus in the ad hoc SINR model. The abstract MAC layer provides basic routines where run-times are bounded by delay-functions. By proving upper bounds on delays in the ad hoc SINR model, they establish an abstract MAC layer that permits single-message broadcast in $\mathcal{O}((D+\log n)\,\text{polylog}\, R)$, which is an improvement over [4].

The algorithm of [4] is the simplest and seemingly most robust of the global broadcast algorithms in the ad hoc SINR model and we therefore decided to extend this algorithm to the adversarial, unreliable SINR model considered in the present paper. However, it would certainly be interesting to see whether the approaches of [11] and [15] can also be adapted to work in a more dynamic and unreliable setting.

Adversarial Models in Wireless Networks. To the best of our knowlege, the only previous paper that analyzes non-deterministic, adversarial behavior in combination with the SINR model, is by Ogierman et al. [23]. They assume that the SINR noise parameter is controlled by an adaptive adversary with a restricted energy budget during each interval of some fixed length. The objective of [23] is to design a MAC layer that allows to achieve optimal throughput. The algorithm has a constant competitive ratio and it is based on a protocol

[2] The algorithm of [4] – which takes $\mathcal{O}(D\log n(\log^\star n + \log^{\alpha+1} R))$ time – is faster than the algorithm of [15] if $R \leq 2^{\log^{1/c} n}$ for a sufficiently large constant $c > \alpha$ (α is the path loss exponent, see Section 3).

designed in [24] for a simpler wireless network model. We note that the algorithm of [23] heavily relies on the assumption that the nodes can use carrier-sensing.

Non-deterministic behaviour in the broader context of distributed networks has been studied in various settings. Of particular interest in the context of the present paper is the dual graph model introduced by [17]. The dual graph model considers graphs $G = (V, E)$ and $G' = (V, E')$ with $E \subseteq E'$, where G is reliable and G' contains unreliable edges, which are controlled dynamically by an adaptive worst-case adversary. A message can only be delivered from sender to receiver if no other neighbor (in the graph of currently active edges) of the receiver sends. Broadcast in this framework was studied in [8, 17, 19].

Another line of work [1, 2, 5, 9, 20, 24] studies scenarios, where an adversary with a limited energy budget per time may jam the single shared channel or a subset of multiple shared channels, to the effect that no transmissions via the jammed channels are possible. However, all of the aforementioned non-SINR models share the assumption that interference is a strictly local and binary concept (a message is received if and only if exactly one node or neighbor sends) and they all fail to capture the global and continuous interference concept of the SINR model, which is more faithful to reality (cf. [21]).

3 Model and Preliminaries

Communication Model. Let V be a set of n point-shaped communication devices (we call them nodes) embedded to distinct points of a metric space (X, d). For $u, v \in V$, let $d(u, v)$ denote the distance between the two points in X to which u and v are embedded. Nodes are reliable and have unique identifiers. Time is divided into synchronous time slots, henceforth also called rounds. In each round, each node can either transmit a message with a fixed transmission power P or it can listen to the channel and thereby receive at most one message. In addition, each node may conduct an arbitrary amount of computations during each round. We do not explicitly restrict the size of messages, however, in addition to the size of the data that needs to be broadcast we require at most $\mathcal{O}(\log n)$ additional bits.

Let $I \subseteq V \setminus \{u, v\}$ be the set of nodes other than $u, v \in V$ that are transmitting during a round. Then v receives a message from u, iff u transmits, v listens, and

$$\mathrm{SINR}(u, v, I) := \frac{P/d(u, v)^\alpha}{N + \sum_{w \in I} P/d(w, v)^\alpha} \geq \beta_v. \tag{1}$$

We call the parameters P, N, α, and β_v in Equation 1 the *SINR parameters*. The value of $P > 0$ denotes the transmission power of all nodes. Further, the parameter $N \geq 0$ describes the background noise and $\alpha \geq 2$ specifies the power loss of the transmitted signals. Finally, for each node $v \in V$, the threshold $\beta_v > 0$ determines how large the signal to interference and noise ratio at node v needs to be, such that v successfully decodes a signal.

In our abstract model, we assume that the SINR parameters P, N, and α are fixed and that only the value of β_v is subject to some variability and uncertainty. More specifically, there are generally known lower and upper bounds $\beta_{\min} \geq 1$ and $\beta_{\max} > \beta_{\min}$ on β_v. In each round and for all nodes $v \in V$, the actual value of β_v is determined by a *strongly adaptive adversary* arbitrarily in the range $[\beta_{\min}, \beta_{\max}]$. That is, the adversary can adjust the value β_v for each node and can thereby control to some extent, whether a given message is received or not. We note that to keep our abstract model as simple as possible, we assume that the adversary can only change β_v while all other SINR parameters are fixed (and globally known). In Section 9, we discuss how our abstract model can be used to capture a more powerful adversary that can vary all SINR parameters within a given range.

Communication Graph. We define the *maximum range* $r_m := (P/N\beta_{\min})^{\frac{1}{\alpha}}$, *maximum safe range* $r_s := (P/N\beta_{\max})^{\frac{1}{\alpha}}$ and the *effective range* $r_e := r_s/(1+\rho)$ for constant $\rho > 0$. For a given $I \subseteq V$ we call a transmission from u to v *safe* if $\mathrm{SINR}(u, v, I) \geq \beta_{\max}$, otherwise the transmission is called *unsafe*. For $d(u, v) > r_m$ no transmissions among u and v are possible. For $r_s < d(u, v) \leq r_m$ only unsafe transmissions are possible. For $d(u, v) \leq r_s$ and especially $d(u, v) \leq r_e$ safe transmissions are possible (e.g., if $I = \emptyset$). Now we are able to formally define the communication graph G_C as $(V, \{\{u, v\} \,|\, u, v \in V, u \neq v, d(u, v) \leq r_e\})$.

All nodes are assumed to know polynomial bounds on the number of nodes n and on the ratio $R := \frac{d_{\max}}{d_{\min}}$ of the maximum distance $d_{\max}$ to the minimum distance $d_{\min}$ among neighbors in the communication graph G_C. The values of n and R appear only inside log functions. Since it makes no difference in the *asymptotic* running time of our algorithm, we will assume exact knowledge of n and R for simplicity. Further, we assume that the network is *growth-bounded* in the following sense. There exists a constant $\delta \in [1, \alpha)$ such that for any subset of nodes $S \subseteq V$, for any $v \in S$ and any $x \in \mathbb{R}^+$, the number of nodes from S that are within distance $x \cdot d^S_{\min}$ of v is bounded by $\mathcal{O}(x^\delta)$, where $d^S_{\min} := \min_{u,v \in S, u \neq v} d(u, v)$.[3]

Global Broadcast and Neighborhood Dissemination. In the present paper, we intend to solve the *global broadcast problem*. There is a distinguished source node s that initially has a broadcast message $\mathcal{M}$. The global broadcast problem is solved when $\mathcal{M}$ is disseminated to all nodes in V. We assume that a node v cannot participate in a global broadcast algorithm before v learns the broadcast message $\mathcal{M}$. This assumption is sometimes known as *asynchronous* start. We are interested in *randomized* algorithms that solve the global broadcast problem *with high probability* (w.h.p.) that is, with probability at least $1 - 1/n^c$ for a given, sufficiently large constant $c > 0$. The algorithms' guarantees have to hold for *any* strategy of the adversary, i.e., for every possible way in which the adversary may choose β_v for each round and node. We call an algorithm that fulfills these criteria *robust*.

Our solution to the global broadcast problem is based on a solution to the following *neighborhood dissemination* problem. Assume that a subset $S \subseteq V$ of the nodes knows some message $\mathcal{M}$. Neighborhood dissemination is solved as soon as all neighbors $N(S)$[4] of S in G_C know $\mathcal{M}$. Note that if D is the diameter of G_C, the global broadcast problem can clearly be solved by running neighborhood dissemination D times.

Spatial Reuse in Dense Networks. In dense networks we can take advantage of the fact that signals fade polynomially with distance. At the same time, the interference from nodes within a distance around v grows not too fast (when said distance is increased) due to the growth bound. These properties potentially allow large numbers of concurrent transmissions in densely packed areas. This is sometimes known as spatial reuse.

The following lemma gives a formalization of this property and is slightly adapted from Daum et al. [4]. Assume that a set of active nodes all transmit with a fixed probability p and that all other nodes are silent. Then there is a *constant* probability that two nodes can communicate safely, given that they are within safe transmission range of each other and are closer than the smallest distance among active nodes times a constant.

---

[3] Intuitively, the growth bound can be interpreted as follows. Assume we grow the radius x of a sphere around a node v. Then the interference caused by nodes at distance x fades faster (i.e. with $1/x^\alpha$) than the number of nodes within that sphere grows (i.e. with $\mathcal{O}(x^\delta)$). This implies that the total interference from distant nodes decreases (the important feature is $\delta < \alpha$), as we show in Lemma 1. This notion is generalized for arbitrary subsets $S \in V$ and 'normalized for granularity' $d^S_{\min}$.

[4] $N(S) := \{v \in V \mid \text{there is } u \in S \text{ such that } (u, v) \text{ is an edge in } G_C\}$

▶ **Lemma 1** (cf. [4]). *Let $S \subseteq V$ be a set of active nodes sending with fixed probability p. All non-transmitting nodes listen. Consider two nodes $u \in S$ and $v \in V$ with $d(u,v) \leq c_1 \cdot r_e$ and $d(u,v) \leq c_2 \cdot d^S_{\min}$, where $0 < c_1 < 1+\rho$, $c_2 > 0$ are constants and $d^S_{\min} = \min_{x \neq y \in S} d(x,y)$. If v listens, then a safe transmission from u to v takes place with constant probability $\mu \in (0,p)$.*

Proof. We show $\mathrm{SINR}(u,v,I) \geq \beta_{\max}$. For this purpose we compute a bound on $I^{\max}_{S'}(v)$ defined as the maximum interference at v originating from $S' \coloneqq \{w \in S \mid d(v,w) \geq k_0 \cdot d^S_{\min}\}$ with $k_0 \in \mathbb{N}$. We partition $S' = \bigcup_{k=k_0}^{\infty} R_k$ along concentric rings $R_k \coloneqq \{w \in S' \mid k \cdot d^S_{\min} \leq d(v,w) < (k+1) \cdot d^S_{\min}\}$ of thickness $d^S_{\min}$. Moreover, let $B_k \coloneqq \{w \in S \mid d(v,w) < (k+1) d^S_{\min}\}$ be the nodes within a ball of radius $(k+1)d^S_{\min}$ centered at v.

First we unwrap the definition of the growth bound $|B_k| \in \mathcal{O}((k+1)^\delta) = \mathcal{O}(k^\delta)$: There are constants $k_0, \zeta > 0$ such that $|B_k| \leq \zeta \cdot k^\delta$ for all $k \geq k_0$. This allows us to bound the interference at v stemming from R_{k_0}. In the worst case (regarding the amount of interference from R_{k_0}) all of the at most $\zeta \cdot k_0^\delta$ nodes in B_{k_0} are located in R_{k_0}, which gives us

$$I^{\max}_{R_{k_0}}(v) = \sum_{w \in R_{k_0}} \frac{P}{d(v,w)^\alpha} \leq \zeta \cdot k_0^{\delta-\alpha} \frac{P}{(d^S_{\min})^\alpha} \in \mathcal{O}(k_0^{\delta-\alpha}) \frac{P}{(d^S_{\min})^\alpha}.$$

Now we bound the combined interference from the other rings R_k, with $k > k_0$. Again, we look at the worst case regarding interference at v. The maximum interference without violating the growth bound is obtained when as many nodes as possible are as close to v as possible. Therefore, B_{k-1} contains as many nodes as the growth bound allows, implying that $|R_k| \leq \zeta k^\delta - |B_{k-1}| \leq \zeta(k^\delta - (k-1)^\delta) \leq \zeta\delta k^{\delta-1}$ (the last inequality is given in Appendix C Lemma 17). This leads us to

$$\begin{aligned}\sum_{k=k_0+1}^{\infty} I^{\max}_{R_k}(v) &\leq \zeta\delta \sum_{k=k_0+1}^{\infty} k^{\delta-\alpha-1} \frac{P}{(d^S_{\min})^\alpha} \\ &\leq \zeta\delta \int_{k=k_0}^{\infty} k^{\delta-\alpha-1} dk \frac{P}{(d^S_{\min})^\alpha} = \zeta\delta \frac{k_0^{\delta-\alpha} P}{(\alpha-\delta)(d^S_{\min})^\alpha} \in \mathcal{O}(k_0^{\delta-\alpha}) \frac{P}{(d^S_{\min})^\alpha}.\end{aligned}$$

Combining these results we observe $I^{\max}_{S'}(v) \leq \kappa(k_0) P / (d^S_{\min})^\alpha$ for a sequence $\kappa(k_0) \in \mathcal{O}(k_0^{\delta-\alpha})$, which approaches 0 for $k_0 \to \infty$. Assuming all nodes in $S \setminus S'$ remain silent we obtain

$$\begin{aligned}\mathrm{SINR}(u,v,S') &= \frac{\frac{P}{d(u,v)^\alpha}}{N + I^{\max}_{S'}(v)} \overset{(*)}{\geq} \frac{P}{\frac{d(u,v)^\alpha P}{r_s^\alpha \beta_{\max}} + \frac{d(u,v)^\alpha \kappa(k_0) P}{(d^S_{\min})^\alpha}} \qquad (*)\colon N = \tfrac{P}{r_s^\alpha \beta_{\max}} \text{ (cf. def. of } r_s) \\ &\geq \frac{P}{\frac{c_1^\alpha r_e^\alpha P}{(1+\rho)^\alpha r_e^\alpha \beta_{\max}} + \frac{(c_2 d^S_{\min})^\alpha \kappa(k_0) P}{(d^S_{\min})^\alpha}} = \frac{\beta_{\max}}{\underbrace{\frac{c_1^\alpha}{(1+\rho)^\alpha}}_{<1} + \underbrace{c_2^\alpha \beta_{\max} \kappa(k_0)}_{\to 0, \text{ for } k_0 \to \infty}}} \geq \beta_{\max}\end{aligned}$$

for sufficiently large $k_0 \in \Theta(1)$.[5] Thus a safe transmission from u to v takes place with probability $\mu \in p(1-p)^{\mathcal{O}(k_0^\delta)}$ defined as the probability that all nodes in $S \setminus S'$ are silent.[6] ◀

[5] Note that k_0 does not depend on the size n, active nodes S, nor on the granularity R of the network. Only on SINR parameters, growth bound exponent δ and c_1, c_2, which we deem all constant.

[6] Daum et al. [4] propose to choose p such that $\mu(p, k_0)$ is maximized.

Algorithm 1 RobustDissemination. ▷ *high level description*

$S_1 \leftarrow S$ ▷ *set of active nodes at start of this procedure*
for phase $\phi \leftarrow 1$ **to** $\lfloor \log R \rfloor + 2$ **do** ▷ *number of phases limited due to Lemma 11*
  for $\Theta(Q \log n)$ rounds **do** ▷ *Q determined in Lemmas 13,14*
    Nodes in S_ϕ send $\mathcal{M}$ with probability p/Q ▷ *disseminate $\mathcal{M}$ to $N(S)$*
  Compute DIS $S_{\phi+1}$ of $\tilde{H}^\mu[S_\phi]$ by executing subroutine ComputeDIS(μ, p) on all $v \in S$

4 Robust Broadcast Algorithm - Overview

The solution of the *neighborhood dissemination problem* lies at the core of the broadcast algorithm by [4]. An algorithm solves neighborhood dissemination for a set of active nodes S that already know a message $\mathcal{M}$, if after its execution, the neighborhood $N(S)$ of S in the communication graph G_C also knows $\mathcal{M}$. Starting with a source node v as single active node $S = \{v\}$, global broadcast can be solved by iteratively calling the neighborhood dissemination routine, setting $S = N(S)$ after each iteration. This way $\mathcal{M}$ travels one hop along all shortest paths in G_C emanating from the source, thus solving global broadcast after at most D calls of the neighborhood dissemination routine (D is the diameter of G_C).

Algorithm RobustDissemination (introduced above) solves the neighborhood dissemination problem in a robust manner. This means that the synchronous execution of RobustDissemination by all active nodes $v \in S$ disseminates message $\mathcal{M}$ to $N(S)$ w.h.p. and for any strategy of the adversary. It differs from [4] only in its subroutines and its analysis.[7] The dissemination of $\mathcal{M}$ from S to $N(S)$, relies on a graph structure $H^\mu[S]$ among the set S of active nodes, named *SINR-induced graph* by [4]. The graph $H^\mu[S]$ contains all edges among nodes in S with a probability of transmission success of at least μ.

In [4], active nodes approximate $H^\mu[S]$ by exchanging messages and computing the ratio of successfully received messages in order to determine reliable links. In the unreliable SINR model, the probability of transmission success is subject to adversary influence. In our analysis we account for that by modifying the definition of $H^\mu[S]$ and classify edges into safe and unsafe edges (implying that $H^\mu[S]$ is not unique, since the adversary may 'choose' unsafe edges). The structure $H^\mu[S]$ itself is implicit. We introduce a subroutine Transmit that, when executed simultaneously by all nodes in S, 'probes' connections and passes messages only among those nodes in S which are sufficiently 'reliable', while prohibiting communication among all others, thereby inducing $H^\mu[S]$. The details are covered in Section 5.

Algorithm RobustDissemination is grouped into phases during which active nodes send. After each phase, active nodes are thinned out to decrease interference and enable dissemination of $\mathcal{M}$ to more distant neighbors. In [4], this is achieved by calculating a maximal independent set (MIS) on $H^\mu[S]$ and deactivating the nodes not in it. Due to unreliable edges we are not able to compute a MIS of $H^\mu[S]$ in our scenario. Instead, we use the well-known coloring algorithm of [18] to construct a structure we call *Dominating Independent Set* (DIS) in $\mathcal{O}(\log n \log^\star n)$ rounds and has properties similar to a MIS (this structure was used before in the context of dual graphs by [3]). The algorithm ComputeDIS uses Transmit as a subroutine and is given and analyzed in Section 6.

We show that after calculating a DIS on S and deactivating all nodes not in it, the minimal distance among remaining nodes in S at least doubles. Hence, after at most $\mathcal{O}(\log R)$ phases their distance exceeds r_e (remaining nodes are *'sparse'*) and by then (at the latest) $\mathcal{M}$

[7] We adjust the proofs of [4] to this paper, thus knwoledge of [4] is recommended but not required. Additionally we mark all lemmas, which have a direct counterpart in [4] with "(cf. [4])".

is received by all nodes in $N(S)$ w.h.p. The properties required for the analysis of Algorithm 1 are given in Section 7 where we adhere closely to the respective proofs provided in [4].[8]

In the following, we present the details and subroutines of RobustDissemination. We show that key characteristics of the algorithm of [4] are conserved or transformed into similar notions in the unreliable case. This enables us to prove the following theorem in Section 8.

▶ **Theorem 2.** *Algorithm* RobustDissemination *solves neighborhood dissemination in the unreliable SINR model robustly and in* $\mathcal{O}(\log n\,(\log^\star n + (\log R)^{\alpha+1}\frac{\beta_{\max}}{\beta_{\min}}))$ *rounds.*

We can solve global broadcast by repeating algorithm RobustDissemination D times, a fact we note in the following corollary.

▶ **Corollary 3.** *Algorithm* RobustDissemination *can be used to solve global broadcast in the unreliable SINR model robustly and in* $\mathcal{O}(D\,\log n\,(\log^\star n + (\log R)^{\alpha+1}\frac{\beta_{\max}}{\beta_{\min}}))$ *rounds.*

5 SINR-Induced Graphs in the Unreliable SINR Model

Assume that during each round the active nodes in S send with the same probability $p\in(0,1)$, which yields a random set $I\subseteq S$ of transmitting nodes. For every pair of nodes u,v let $\sigma_{u,v}$ be the probability that $\mathrm{SINR}(u,v,I)\geq\beta_{\max}$, i.e., the probability that a safe transmission from u to v takes place. Let $\tau_{u,v}$ be the probability that $\mathrm{SINR}(u,v,I)\geq\beta_{\min}$, i.e., an unsafe transmission *may* take place (the adversary might decide). Obviously $\tau_{u,v}\geq\sigma_{u,v}$.

Let $\mu\in(0,p)$ be a given threshold probability. Then we call v a *μ-safe neighbor* of u and (u,v) a *μ-safe edge* if both $\sigma_{u,v}\geq\mu$ and $\sigma_{v,u}\geq\mu$. We highlight the fact that (u,v) is μ-safe iff (v,u) is μ-safe as well, with the notation $\{u,v\}$. We call v a *μ-unsafe neighbor* of u and (u,v) a *μ-unsafe edge* if $\tau_{u,v}\geq\mu$ and (u,v) is not μ-safe.

Define the (directed) *SINR-induced graph* $H^\mu[S]:=(S,E^\mu[S])$ where $(u,v)\in E^\mu[S]$, iff $\tau_{u,v}\geq\mu$. This means $E^\mu[S]$ contains exactly the μ-safe and μ-unsafe edges. Let $E^\mu_{\text{safe}}[S]\cup E^\mu_{\text{unsafe}}[S]=E^\mu[S]$ with $E^\mu_{\text{safe}}[S]\cap E^\mu_{\text{unsafe}}[S]=\emptyset$ be the partition of $E^\mu[S]$ into μ-safe edges and μ-unsafe edges.

We cannot hope to determine $H^\mu[S]$ efficiently and exactly with a distributed algorithm in the unreliable scenario, since the adversary might mask the existence of μ-unsafe edges. Instead we settle for an ε-close approximation $\tilde{H}^\mu[S]$ (for an $\varepsilon\in(0,\frac{1}{2}]$).[9] A graph $\tilde{H}^\mu[S]:=(S,\tilde{E}^\mu[S])$ is an ε-close approximation of $H^\mu[S]$ if and only if

$$E^\mu_{\text{safe}}[S]\subseteq\tilde{E}^\mu[S]\subseteq E^{(1-\varepsilon)\mu}[S].$$

This entails that $\tilde{H}^\mu[S]$ guarantees to include μ-safe edges with $\sigma_{u,v},\sigma_{v,u}\geq\mu$ and to exclude edges with $\tau_{u,v}<(1-\varepsilon)\mu$. A graph $\tilde{H}^\mu[S]$ may or may not include edges (u,v) with $\tau_{u,v}\geq(1-\varepsilon)\mu$ and $\sigma_{u,v}<\mu$ or $\sigma_{v,u}<\mu$, though. Due to these edges, an approximation $\tilde{H}^\mu[S]$ is not unique and communication in $\tilde{H}^\mu[S]$ along these edges is inherently volatile.

Note that instead of computing $\tilde{H}^\mu[S]$ (as in [4]), we utilize that communication via Transmit(μ,p,M) (Algorithm 2) induces a graph $\tilde{H}^\mu[S]$ by guaranteeing (w.h.p.) communication along edges in $E^\mu_{\text{safe}}[S]$ and inhibiting communication among edges (u,v) with $\tau_{u,v}<(1-\varepsilon)\mu$ w.h.p. (the adversary decides about the remaining edges in $E^{(1-\varepsilon)\mu}[S]$). We

[8] Algorithm RobustDissemination uses the following constants: Network param. n,R, SINR param. $P,N,\alpha,\beta_{\min},\beta_{\max}$, and tuning param. p,ε. Parameter μ is derived from these constants (cf. Lemma 1).

[9] $\varepsilon=\frac{1}{2}$ optimizes the run-time of Transmit$(\mu,p,\mathcal{M}_v)$. The increased in-degree Δ of $\tilde{H}^\mu[S]$ associated with this choice of ε is of comparatively little consequence regarding run-time.

prove this in Lemma 5. The properties of $\tilde{H}^\mu[S]$ are leveraged in the analysis of the DIS (in whose computation we actually employ $\textsc{Transmit}(\mu, p, M)$, cf. Section 6) and in the analysis of the neighborhood dissemination protocol (Section 7).

A convenient property of $\tilde{H}^\mu[S]$ we immediately observe, is that its maximum in-degree is bounded by $\Delta := \frac{1}{(1-\varepsilon)\mu}$, since a higher in-degree implies a node v that might receive $\sum_{u\in N(v)} \tau_{u,v} \geq \sum_{u\in N(v)} (1-\varepsilon)\mu > 1$ messages in expectation, contradicting the fact that nodes receive at most one message per round. Another property is shown by the subsequent lemma, namely that there is a small enough constant μ, such that $H^\mu[S]$ contains all relatively short edges as μ-safe edges. Since $E^\mu_{\text{safe}}[S] \subseteq \tilde{E}^\mu[S]$, this lemma extends to $\tilde{H}^\mu[S]$.

▶ **Lemma 4** (cf. [4]). *There exists $\mu \in (0,p)$, s.t. for any $S \subseteq V$ and all $u, v \in S, u \neq v$ with $d(u,v) \leq \min(2d^S_{\min}, r_e)$, $d^S_{\min} = \min_{u\neq v\in S} d(u,v)$, it holds that $\{u,v\} \in E^\mu_{safe}[S]$.*

Proof. Since $d(u,v) \leq 2 \cdot d^S_{\min}$ and $d(u,v) \leq r_e$, this lemma is a corollary of Lemma 1. ◀

The following routine guarantees safe and fast message passing among μ-safe neighbors in S w.h.p. Furthermore, it inhibits communication among nodes in S with low transmission probability w.h.p. Edges along which communication takes place form a graph $\tilde{H}^\mu[S]$.

Algorithm 2 $\textsc{Transmit}(\mu, p, \mathcal{M}_v)$. ▷ *to be initiated simultaneously by each $v \in S$*

for $T \leftarrow \frac{c \log n}{\varepsilon^2 \mu}$ rounds **do** ▷ *c determined in Lemma 5 and $\varepsilon \in (0, \frac{1}{2})$ fixed*
  Send pair $(\text{ID}(v), \mathcal{M}_v)$ of message $\mathcal{M}_v$ and own $\text{ID}(v)$ with probability p
for T rounds **do** ▷ *list length constant, since nodes have at most $\frac{1}{(1-\varepsilon)\mu}$ neighbors*
  With probability p, send list of IDs of which at least $(1-\frac{\varepsilon}{2})\mu T$ messages were received
for all $\text{ID}(u)$ in own ID-list **do**
  if $\text{ID}(u)$ is in own list **and** $\text{ID}(v)$ is in the ID-list received from u **then**
    Consider message $\mathcal{M}_u$ of u as received ▷ *receive messages where $\sigma_{u,v}, \sigma_{v,u} \geq \mu$*
  else discard $\mathcal{M}_u$ ▷ *neglect messages where $\tau_{u,v} < (1-\varepsilon)\mu$*

▶ **Lemma 5.** *Algorithm $\textsc{Transmit}(\mu, p, \mathcal{M}_v)$ takes $\mathcal{O}(\frac{\log n}{\varepsilon^2 \mu})$ rounds. If initiated simultaneously by each $v \in S$, then messages are received among μ-safe neighbors u, v ($\sigma_{u,v} \geq \mu$) w.h.p. If $\tau_{u,v} < (1-\varepsilon)\mu$ then w.h.p. no message of u is received by v. Furthermore, edges along which messages are successfully received form an ε-close approximation $\tilde{H}^\mu[S]$ of $H^\mu[S]$ w.h.p.*

Proof. Let $u, v \in S$ and let $X_i := 1$, if $\text{SINR}(u, v, I) \geq \beta_{\max}$ in round i of the first loop, i.e., a safe transmission from u to v takes place, and $X_i := 0$, else. Furthermore, let $X := \sum_{i=1}^T X_i$. Assume (u, v) is μ-safe, i.e., $\sigma_{u,v}, \sigma_{v,u} \geq \mu$. Using a multiplicative Chernoff bound we show that the event $X \leq (1 - \frac{\varepsilon}{2})\,\mu T$ is very improbable.

$$\mathbb{P}(X \leq (1 - \frac{\varepsilon}{2})\,\sigma_{u,v} T) \leq \exp(-\frac{\varepsilon^2 \sigma_{u,v} T}{8}) = \exp(-\log n \cdot \frac{c\,\sigma_{u,v}}{8\mu}) \leq n^{-\frac{c}{8}}.$$

Analogously, the same result holds for messages sent from v to u. Let $A_{u,v}$ be the event that v does not receive message $\mathcal{M}_u$ from u via algorithm $\textsc{Transmit}(\mu, p, \mathcal{M}_v)$ that is $X \leq (1 - \frac{\varepsilon}{2})\,\mu T$ during the first loop or u does not receive v's list during the second loop. The second condition is even less likely than the first, hence $\mathbb{P}(A_{u,v}) \leq 2n^{-\frac{c}{8}} \leq n^{1-\frac{c}{8}}$.

This means that (for $c > 8$) event $\bar{A}_{u,v}$ takes place w.h.p., i.e., v receives message $\mathcal{M}_u$ from u. The same is true for event $\bar{A}_{v,u}$. If both $\bar{A}_{u,v}$ and $\bar{A}_{v,u}$ occur, we have $\{u, v\} \in \tilde{E}^\mu_{\text{safe}}[S]$ in the graph $\tilde{H}^\mu[S]$ induced by those edges along which messages are successfully received.

Now let $Y_i := 1$, if $\mathrm{SINR}(u, v, I) \geq \beta_{\min}$ in round i of the first loop, i.e., an unsafe transmission from u to v might take place, and $Y_i := 0$, else. Let $Y := \sum_{i=1}^{T} Y_i$. Assume $\tau_{u,v} < \mu(1-\varepsilon) =: \mu'$ and let $B_{u,v}$ be the event $Y \geq (1-\frac{\varepsilon}{2})\,\mu T$. We obtain

$$\mathbb{P}(B_{u,v}) = \mathbb{P}(Y \geq (1-\frac{\varepsilon}{2})\mu T) = \mathbb{P}(Y \geq \frac{1-\frac{\varepsilon}{2}}{1-\varepsilon}\mu' T) \leq \mathbb{P}(Y \geq (1+\frac{\varepsilon}{2})\,\mu' T)$$
$$\leq \exp(-\frac{\varepsilon^2 \mu' T}{12}) = \exp(-\log n \cdot \frac{c\,\mu'}{12\mu}) = n^{-\frac{c}{12}(1-\varepsilon)} \overset{(\varepsilon \leq \frac{1}{2})}{\leq} n^{-\frac{c}{24}}.$$

Therefore, even if the adversary permits all transmissions with $\mathrm{SINR}(u, v, I) \geq \beta_{\min}$ during the first loop, the total number will be lower than $(1-\frac{\varepsilon}{2})\,\mu T$ w.h.p. Thus u is not in v's list, hence neither receives the others message thus $(u, v), (v, u) \notin \tilde{E}^{\mu}[S]$ w.h.p. The probability that at least one edge of the clique among active nodes $C(S) := \{(u, v) \mid u, v \in S, u \neq v\}$ is falsely in- or excluded in $\tilde{E}^{\mu}[S]$ is bounded by

$$\mathbb{P}(\bigcup_{(u,v)\in C(S)} (A_{u,v} \cup B_{u,v})) \leq \sum_{(u,v)\in C(S)} (\mathbb{P}(A_{u,v}) + \mathbb{P}(B_{u,v})) \leq \sum_{(u,v)\in C(S)} (n^{1-c/8} + n^{-c/24}) < n^{4-c/24}.$$

This shows that for $c > 96$ w.h.p. all μ-safe edges are included in $\tilde{E}^{\mu}[S]$ and all edges which are not even $\mu(1-\varepsilon)$-unsafe are excluded w.h.p. ◀

Note that the definition of $\tilde{H}^{\mu}[S]$ permits directed edges $(u, v) \in \tilde{E}^{\mu}[S]$ with $(v, u) \notin \tilde{E}^{\mu}[S]$. In practice, this happens if $\tau_{u,v}, \tau_{v,u} \geq \mu(1-\varepsilon)$ and the adversary permits all transmissions among u and v in the first loop of Algorithm 2 such that u and v are in each others lists. Subsequently, the adversary allows only the transfer of v's list to u but blocks all attempts of u sending its list to v. Nevertheless, μ-safe edges are undirected w.h.p., since there will occur sufficiently many safe transmissions between μ-safe neighbors, which the adversary cannot block. In the following we use the notation $\{u, v\}$ to highlight μ-safe edges.

6 Calculating Dominating Independent Sets

After each phase of message dissemination of nodes in S, we rely on the properties of $\tilde{H}^{\mu}[S]$, to thin out the set of nodes S so that message dissemination becomes feasible in areas of high interference. In the basic algorithm [4] this is accomplished by determining a MIS of $\tilde{H}^{\mu}[S]$ and deactivating all nodes not in it. In unreliable scenarios we cannot guarantee independence with respect to μ-unsafe edges since the adversary might mask their existence during MIS-calculation. To loosen the requirements, we introduce a more general notion.

▶ **Definition 6.** Let $G = (V, E, E')$ be a graph with disjoint sets of undirected edges E and directed edges E'. Subset $V' \subseteq V$ is independent w.r.t. E if no two nodes in V' are connected by an edge $\{u, v\} \in E$. Subset $V' \subseteq V$ is dominating w.r.t. $E \cup E'$ if for every node $v \in V$ either $v \in V'$ or there exists a $u \in V'$ such that $(u, v) \in E \cup E'$. A *Dominating Independent Set* (DIS) $D \subseteq V$ of G is independent w.r.t. E and dominating w.r.t. $E \cup E'$.

We exploit the fact that the in-degree of any valid graph $\tilde{H}^{\mu}[S]$ is bounded by $\Delta \in \mathcal{O}(1)$ to calculate a DIS of $(S, E^{\mu}_{\mathrm{safe}}[S], E^{(1-\varepsilon)\mu}_{\mathrm{unsafe}}[S])$. First, we determine a $(3\Delta \log \Delta)^2$-coloring of S with respect to the μ-safe edges $E^{\mu}_{\mathrm{safe}}[S]$ in $\mathcal{O}(\log n \log^{\star} n)$ rounds by adapting the method of Linial [18], which uses Δ-cover-free families. Second, we calculate a DIS based on this coloring by successively adding nodes of one color while sustaining the DIS condition.

▶ **Definition 7.** A Δ-cover-free family $\mathcal{F}$ is a family of subsets of $[m]$,[10] such that for any selection of distinct sets $F_0, \dots, F_\Delta \in \mathcal{F}$ it holds that $F_0 \not\subseteq \bigcup_{j=1}^{\Delta} F_j$.

Erdős et al. show in [6] how such families can be constructed using polynomials over finite fields. The proof of the following lemma provides the according construction procedure and is given in Appendix A.

▶ **Lemma 8.** *For prime power q, and integer $d \in [q-1]$ there is a $\lfloor (q-1)/d \rfloor$-cover-free family $\mathcal{F}$ of size $|\mathcal{F}| = q^{d+1}$ of subsets of $[q^2]$.*

From Lemma 8 we immediately obtain the following Lemma 9, which was suggested by Linial in [18] as constructive alternative to his non-constructive proof of an even smaller Δ-cover-free family. For completeness, the lemma is derived in Appendix A.

▶ **Lemma 9.** *For $k \in \mathbb{N}$, there is a Δ-cover-free family $\mathcal{F}$ of subsets of $[m]$ with $|\mathcal{F}| \geq k$ and $m \leq (3\Delta \log k)^2$.*

Algorithm 3 ComputeDIS(μ, p). ▷ *to be initiated simultaneously by each $v \in S$*

$k_0 \leftarrow \mathrm{ID}(v)$ ▷ *assign colors, initially unique IDs*
$k \leftarrow n$ ▷ *k tracks upper bound of colors currently used*
for $\mathcal{O}(\log^\star n)$ times **do** ▷ *loop computes coloring w.r.t. μ-safe edges*
  Transmit(μ, p, k_0) ▷ *send own color to neighbors in $\tilde{H}^\mu[S]$*
  Let $k_1, \dots, k_t$ be the colors received from v's neighbors in $\tilde{H}^\mu[S]$. ▷ *$t \leq \Delta$, $k_i \leq k$*
  Let $\mathcal{F} = \{F_1, \dots, F_k\}$ as in Lemma 9 ▷ *construction procedure in proof of Lemma 8*
  Choose new color $k_0 \in F_{k_0} \setminus \bigcup_{k_i \neq k_0} F_{k_i}$ ▷ *new coloring is valid w.r.t. μ-safe edges*
  $k \leftarrow (3\Delta \log k)^2$ ▷ *at most $(3\Delta \log k)^2$ colors remaining (Lemma 9)*
for $l \leftarrow 1$ **to** k **do** ▷ *loop uses $\mathcal{O}(1)$-coloring to compute DIS*
  if $l = k_0$ **then**
    Transmit$(\mu, p,$'*Do not join DIS and terminate!*'$)$ ▷ *inform neighbors*
    Join DIS and terminate
  else Transmit$(\mu, p, \emptyset)$ ▷ *All active nodes must initiate routine (Lemma 5)*
  if received message '*Do not join DIS and terminate!*' **then** terminate

▶ **Lemma 10.** *Algorithm* ComputeDIS(μ, p) *computes a DIS of $(S, E^\mu_{safe}[S], E^{(1-\varepsilon)\mu}_{unsafe}[S])$ in $\mathcal{O}(\frac{\log n}{\varepsilon^2 \mu} \log^\star n)$ rounds in a robust manner, when initiated simultaneously by all nodes in S.*

Proof. We prove that during the **first loop** of ComputeDIS(μ, p) a valid coloring w.r.t. μ-safe edges is maintained. Initially we have the trivial coloring by IDs. The rest can be shown inductively. Presume that, at the beginning of a given loop cycle, we have a valid coloring with respect to $E^\mu_{\mathrm{safe}}[S]$ of size at most k.

Due to the synchronous execution of Transmit(μ, p, k_v) by each node $v \in S$ (where k_v is the current color of v) every node receives at most $t \leq \Delta$ colors $k_1, \dots, k_t \leq k$ from its neighbors w.h.p., since messages are only transmitted along edges of $\tilde{H}^\mu[S]$ (Lemma 5) whose in-degree is bounded by Δ. In particular, Transmit(μ, p, k_v) guarantees that every node receives the colors from all of its μ-safe neighbors w.h.p. (Lemma 5).

[10] For natural numbers $n \geq 1$ we define $[n] := \{1, \dots, n\}$.

Note that v may also receive colors that conflict with its own, which stem from $\mu(1-\varepsilon)$-unsafe neighbors which are not μ-safe and whose messages were blocked by the adversary during earlier loop cycles. However, since we only need to ensure a valid coloring with respect to μ-safe edges, this does not concern us.

Now all nodes in S construct the same Δ-cover-free family $\mathcal{F} = \{F_1, \dots, F_k\}$ of subsets of $[m]$ with $m \leq (3\Delta \log k)^2$, in accordance with Lemmas 8, 9.[11] Each node $v \in S$ picks a new color $k_0 \in F_{k_0} \setminus \bigcup_{k_i \neq k_0} F_{k_i}$, which exists due to the Δ-cover-free property. Therefore, neighbors that had different colors during previous rounds are (w.h.p.) still differently colored. In particular, this implies that w.h.p. μ-safe neighbors remain differently colored.

In the **second loop** we use the coloring to determine a DIS of S. Let D_l be the subset of nodes in S that have joined the DIS and let T_l be those that have terminated until (and including) loop cycle l. W.h.p., ComputeDIS maintains the invariant that D_l is a DIS of T_l, i.e., $D_l \subseteq T_l$ is independent w.r.t. μ-safe edges and D_l dominates T_l w.r.t. μ-safe edges *and* $\mu(1-\varepsilon)$-unsafe edges.

For $l = 0$, this is obvious since $T_0 = D_0 = \emptyset$. Presume that D_{l-1} is a DIS w.r.t. T_{l-1} and let $v \in T_l \setminus T_{l-1}$. Since μ-safe neighbors are w.h.p. differently colored, $v \in D_l \setminus D_{l-1}$ cannot have a μ-safe neighbor in $D_l \setminus D_{l-1}$ (w.h.p.). Moreover, since D_{l-1} informed its μ-safe neighbors w.h.p. (Lemma 5), they already terminated during previous cycles. Therefore a node $v \in D_l \setminus D_{l-1}$ that joined the DIS during cycle l and terminated only then, cannot have any μ-safe neighbors in D_{l-1} (w.h.p.).

We already know that D_{l-1} dominates T_{l-1} w.r.t. μ-safe edges *and* $\mu(1-\varepsilon)$-unsafe edges (w.h.p.). Every node $v \in T_l \setminus T_{l-1}$ is either in $D_l \setminus D_{l-1}$ or was informed by a neighbor $u \in D_l \setminus D_{l-1}$ (possibly via a $\mu(1-\varepsilon)$-unsafe edge), which means v is dominated by $u \in D_l \setminus D_{l-1}$. Therefore $D_l \setminus D_{l-1}$ dominates $T_l \setminus T_{l-1}$, thus D_l dominates T_l (w.h.p.).

The first loop has the claimed runtime (cf. Lemma 5). The run-time of the second loop depends on the size of the coloring. After at most $\mathcal{O}(\log^\star n)$ cycles of the first loop there are $\mathcal{O}((\Delta \log \Delta)^2) = \mathcal{O}(1)$ colors left. Therefore the run-time of the second loop is dominated by the run-time of the first, proving the claimed overall run-time. ◀

7 Neighborhood Dissemination Analysis

In this section we apply moderate changes to the technical parts given in [4] in order to reuse them in our unreliable SINR model. Let ϕ be a phase of Algorithm 1: RobustDissemination, let S_ϕ be the set of active nodes during phase ϕ and let $d_\phi := \min_{u \neq v \in S_\phi} d(u,v)$, if $|S_\phi| \geq 2$ and $d_\phi := \infty$, else. The following lemma proves that in a DIS $S_{\phi+1}$ of S_ϕ[12] the minimum distance $d_{\phi+1}$ among nodes doubles, or is already larger than the effective communication range r_e. The proof is provided in Appendix B.

▶ **Lemma 11** (cf. [4]). *Let ϕ be a phase of Algorithm 1. There exists a constant $\mu \in (0,p)$, such that $d_\phi \geq 2^{\phi-1} d_{\min}$ or $d_\phi > r_e$, where $d_{\min} = \min_{u \neq v \in V} d(u,v)$.*

The next lemmas show that node $v \in N(S)$ receives the broadcast message $\mathcal{M}$ with a guaranteed probability from its nearest active neighbor $u \in S_\phi$, in case certain conditions are met. These conditions include that u is in safe communication range and that there exists a neighboring node $w \in S_\phi$ of u in $H^\mu[S_\phi]$, which is not too close. With the existence of a

[11] According to our model, nodes have unlimited computing power during each round. Nevertheless Δ-cover-free families can be computed within polynomial time using the constructive proof of Lemma 8.

[12] We presume that μ, ε are fixed and abbreviate 'DIS of $(S, E^\mu_{\text{safe}}[S], E^{(1-\varepsilon)\mu}_{\text{unsafe}}[S])$' with 'DIS of S'.

relatively long edge $(w,u) \in E^\mu[S_\phi]$ we can bound the interference at u using the knowledge that w was able to transfer messages to u with a certain probability $(\tau_{w,v} \geq (1-\varepsilon)\mu)$. Consequently, we are able to limit the interference at u's (close) neighbor v. The proofs of the following lemmas are given in Appendix B. We adhere closely to the proofs provided in [4], although some adaptions and clarifications are required.

First, we limit the interference stemming from the set of nodes close to v and then from the set of distant nodes. For this purpose, let $V' \subseteq V$ be a subset of nodes and let $X^q_{V'}$ be the random number of nodes in V' that send, when each has individual sending probability q. Moreover, let $I^q_{V'}(y) \coloneqq \sum_{x \in V' \text{ sends}} P/d(x,y)^\alpha$ be the random interference at $y \in V$ stemming from sending nodes in V', given that nodes send with probability q. Then $I^{\max}_{V'}(y) \coloneqq \sum_{x \in V'} P/d(x,y)^\alpha$ defines the maximum interference at $y \in V$ from nodes in V'.

▶ **Lemma 12** (cf. [4]). *Let ϕ be a phase of Algorithm 1. Let $v \in N(S)$ and let $u \in S_\phi$ be the node closest to v. Presume there exists $w \in S_\phi$ with an edge $(w,u) \in E^\mu[S_\phi]$ and let w be the farthest such neighbor of u. Let $A = \{x \in S_\phi \mid d(u,x) \leq 2d(u,w)\} \setminus \{u,v\}$ and $\bar{A} = S_\phi \setminus (A \cup \{u,v\})$. If active nodes send with probability p/Q, then there exists $\eta \in \Theta(\beta_{\min})$, such that the following events occur simultaneously with probability at least $\frac{(1-\varepsilon)\mu p}{8Q} \in \Theta(1/Q)$:*

(i) $I^{p/Q}_A(v) \leq \frac{2^{\alpha+1}P}{Q\beta_{\min} d(u,v)^\alpha}$, (ii) $I^{p/Q}_{\bar{A}}(v) \leq \frac{2^{\alpha+1}\eta P}{Q\beta_{\min} d(u,w)^\alpha}$, (iii) u sends and v listens.

Lemma 13 proves that there is a constant probability that $v \in N(S)$ receives $\mathcal{M}$ from u, in case v is in communication range of u and the ratio $d(u,w)/d(u,v)$ is above a threshold.

▶ **Lemma 13** (cf. [4]). *Let ϕ be a phase of Algorithm 1. Let $v \in N(S)$ and $u, w \in S_\phi$ as in Lemma 12. Then there exists $\hat{Q} \in \mathcal{O}(\frac{\beta_{\max}}{\beta_{\min}} 2^\alpha), \gamma \in \Theta((\frac{\beta_{\max}}{\beta_{\min}})^{1/\alpha})$ such that for all $Q \geq \hat{Q}$, node v receives $\mathcal{M}$ safely with probability $\Theta(1/Q)$, if active nodes send with probability p/Q and $d(u,v) \leq (1+\frac{\rho}{2})r_e$ and $d(u,w) \geq \gamma Q^{-\alpha} d(u,v)$.*

Lemma 14 guarantees for each phase ϕ except the last that $v \in N(S)$ receives $\mathcal{M}$ w.h.p. in this or previous phases, or v is still in safe communication range of an active neighbor.

▶ **Lemma 14** (cf. [4]). *Let $\phi \leq \log R + 1$ be a phase of Algorithm 1. Let $v \in N(S)$ and let $u_\phi \in S_\phi$ be the active node closest to v. Then there exists a $Q \in \Theta((\log R)^\alpha), Q \geq \hat{Q}$ such that either v receives $\mathcal{M}$ w.h.p. during phase ϕ or earlier, or $d(u_{\phi+1}, v) \leq r_e(1+\frac{\rho\phi}{2\log R})$.*

Finally, we show that if S_ϕ with $|S_\phi| \geq 2$ is *'sparse'* in the sense that $d_\phi > r_e$ (which is a typical case in at least one phase of Algorithm 1), then $v \in N(S)$ receives $\mathcal{M}$ from $u_\phi \in S_\phi$.

▶ **Lemma 15.** *Let ϕ be a phase of Algorithm 1. Let $|S_\phi| \geq 2$ and $d_\phi > r_e$. If for $v \in V$ there is a node $u \in S_\phi$ with $d(u,v) < c_1 \cdot r_e$ with $0 < c_1 < 1+\rho$, then v receives $\mathcal{M}$ in phase ϕ w.h.p.*

8 Proof of Theorem 2

Proof. Using the previous lemmas, we show that algorithm RobustDissemination is correct and has the claimed run-time. For the correctness, we prove that any node $v \in N(S)$ receives $\mathcal{M}$ w.h.p. during the last phase $\psi \coloneqq \lfloor \log R \rfloor + 2$, at the latest.

Lemma 11 shows that for the minimum distance $d_\phi = \min_{u \neq v \in S_\phi} d(u,v)$ among nodes in S_ϕ, it holds that $d_\phi \geq 2^{\phi-1} d_{\min}$ or $d_\phi > r_e$. Hence, if $\phi > \log R + 1$, then $2^{\phi-1} d_{\min} > R \cdot d_{\min} = d_{\max}$. Therefore, in the last, ψ-th phase, we have $d_\psi \geq 2^{\psi-1} d_{\min} > d_{\max}$ and consequently $d_\psi > r_e$ (recall that $d_{\max}$ is the maximum distance among any two nodes within range r_e of each other). Thus, during the last phase ψ, either $d_\psi = \infty$ in case $|S_\psi| = 1$, or the remaining nodes in S_ψ are *sparse*, in the sense that their minimum distance is greater than r_e.

First we approach the case $|S_\psi|=1$. For this purpose, let ϕ be the first phase for which only one node remains in S_ϕ, i.e., either $|S_1| = 1$ or $|S_{\phi-1}| \geq 2$. In case $S_1 = \{u\}$, node $v \in N(S_1)$ receives $\mathcal{M}$ safely and w.h.p., during the inner loop where we disseminate $\mathcal{M}$ for $\Theta(Q \log n)$ rounds with probability p/Q. This is due to the fact that $d(v,u) \leq r_e$ $(v \in N(S))$ and there is no interference from any other node. Additionally, we choose the hidden constant in $\Theta(Q \log n)$ sufficiently large, such that u sends at least once during that loop w.h.p.

Otherwise ($|S_{\phi-1}| \geq 2$), we apply Lemma 14 on phase $(\phi-1)$, and see that either v has received $\mathcal{M}$ already (in that case we are done) or $d(u_\phi, v) \leq r_e(1 + \frac{\rho(\phi-1)}{2 \log R}) < r_e(1+\rho) = r_s$. The latter condition implies that v is in safe communication range of the only active node u_ϕ in phase ϕ, hence v receives $\mathcal{M}$ w.h.p. during round ϕ using the same argument as before.

Now consider $|S_\psi| \geq 2$. Then Lemma 14 applies for phase $(\psi-1) \leq \log R + 1$, therefore either v already received $\mathcal{M}$ w.h.p. or $d(u_\psi, v) \leq r_e(1 + \frac{\rho(\psi-1)}{2 \log R}) \leq c_1 \cdot r_e$ with $c_1 \leq 1 + \frac{3}{4}\rho$.[13] Since in the final phase ψ, S_ψ is also sparse in the sense that $d_\psi > r_e$, the premise of Lemma 15 is fulfilled, thus v receives $\mathcal{M}$ from its closest neighbor u_ψ in S_ψ w.h.p.

During each phase we execute the sub-procedure ComputeDIS, which takes $\mathcal{O}(\log n \log^\star n)$ rounds (Lemma 10). Sending $\mathcal{M}$ takes $\mathcal{O}(Q \log n) \subseteq \mathcal{O}(\frac{\beta_{\max}}{\beta_{\min}} (\log R)^\alpha \log n)$ rounds each phase (we determined Q in Lemma 14). We have $\lfloor \log R \rfloor + 2$ phases, therefore algorithm Robust-Dissemination takes at most $\mathcal{O}(\log n \, (\log^\star n + (\log R)^{\alpha+1} \frac{\beta_{\max}}{\beta_{\min}}))$ rounds. ◀

9 Adversary Reduction

Finally, we show that a seemingly stronger adversary, which controls *all* SINR-parameters can be reduced to our abstract model, where it controls only β_v. Assume that nodes know only upper and lower bounds $P^\uparrow > P^\downarrow > 0, N^\uparrow > N^\downarrow > 0, \alpha^\uparrow > \alpha^\downarrow \geq 1$, and $\beta^\uparrow > \beta^\downarrow \geq 1$ on the true SINR parameters $P_v, N_v, \alpha(u,v), \beta_v$. In each round and for all pairs of nodes $u, v \in V$, the adversary determines the actual values P_v, N_v, $\alpha(u,v)$, and β_v arbitrarily within the given upper and lower bounds, thereby influencing the outcome of

$$\mathrm{SINR}(u,v,I) \coloneqq \frac{P_u / d(u,v)^{\alpha(u,v)}}{N_v + \sum_{w \in I} P_w / d(w,v)^{\alpha(w,v)}} \geq \beta_v.$$

The following theorem formalizes the fact that instead of picking values for all SINR parameters $P_v, N_v, \alpha(u,v), \beta_v$ within the given upper and lower bounds, the adversary can equivalently modify only β'_v within some enlarged interval $[\beta_{\min}, \beta_{\max}]$ with $\beta_{\min} \leq \beta^\downarrow < \beta^\uparrow \leq \beta_{\max}$, while the other SINR parameters remain static (and globally known).

▶ **Theorem 16.** *For sufficiently large $\beta^\downarrow$ and a fixed set of uniform SINR parameters $P \in [P^\downarrow, P^\uparrow]$, $N \in [N^\downarrow, N^\uparrow]$, $\alpha \in [\alpha^\downarrow, \alpha^\uparrow]$, and $\beta \in [\beta^\downarrow, \beta^\uparrow]$, there are values $\beta_{\max} \geq \beta_{\min} \geq 1$ such that for any choice of $P_v, N_v, \alpha(u,v), \beta_v$ within the bounds, there is a $\beta'_v \in [\beta_{\min}, \beta_{\max}]$ s.t.*

$$\frac{P_u / d(u,v)^{\alpha(u,v)}}{N_v + \sum_{w \in I} P_w / d(w,v)^{\alpha(w,v)}} \geq \beta_v \quad \Longleftrightarrow \quad \frac{P / d(u,v)^{\alpha}}{N + \sum_{w \in I} P / d(w,v)^{\alpha}} \geq \beta'_v. \tag{2}$$

Proof. For brevity, let $\mathrm{SINR}_{\mathrm{adv}}(u,v,I)$ be the formula with the *adversary* controlled parameters P_v, N_v, $\alpha(u,v)$ (left-hand side of (2)) and analogously let $\mathrm{SINR}_{\mathrm{uni}}(u,v,I)$ be defined with the *uniform* parameters P, N, α (right-hand side of (2)). For a given $I \subseteq V \setminus \{u,v\}$, we interpret $C_{u,v,I} \coloneqq \mathrm{SINR}_{\mathrm{adv}}(u,v,I) - \beta_v$ as *connection strength* from u to v. Recall that v receives a message from u if and only if $C_{u,v,I} \geq 0$. Similarly we define $C^{\mathrm{uni}}_{u,v,I} \coloneqq \mathrm{SINR}_{\mathrm{uni}}(u,v,I) - \beta$.

[13] We assume $\log R \geq 2$. Otherwise the neighborhood dissemination can easily be solved with Lemma 1.

Instead of going through the ramifications of the functional dependencies among the SINR parameters occurring in $C_{u,v,I}$, we acknowledge that for the given bounds on the SINR parameters and fixed I, the connection strength $C_{u,v,I}$, treated as function of the SINR parameters, does not diverge. Hence there is a fixed range $[C^{\downarrow}_{u,v,I}, C^{\uparrow}_{u,v,I}]$, from which the adversary chooses $C_{u,v,I}$. We define[14] the *largest negative deviation* Δ^- and the *largest positive deviation* Δ^+ of $C_{u,v,I}$ from $C^{\text{uni}}_{u,v,I}$

$$\Delta^- := \min_{u\neq v\in V,\, I\subseteq V}\left(C^{\downarrow}_{u,v,I} - C^{\text{uni}}_{u,v,I}\right) \leq 0, \quad \Delta^+ := \max_{u\neq v\in V,\, I\subseteq V}\left(C^{\uparrow}_{u,v,I} - C^{\text{uni}}_{u,v,I}\right) \geq 0.$$

Suppose the adversary influences the deviation of $C_{u,v,I}$ from $C^{\text{uni}}_{u,v,I}$ via a *deviation variable* $\Delta \in [\Delta^-, \Delta^+]$, then

$$\begin{aligned}\text{SINR}_{\text{adv}}(u,v,I) \geq \beta_v \iff C_{u,v,I} \geq 0 \iff C^{\text{uni}}_{u,v,I} + \Delta \geq 0\\ \iff \text{SINR}_{\text{uni}}(u,v,I) \geq \beta - \Delta \iff \text{SINR}_{\text{uni}}(u,v,I) \geq \beta'_v,\end{aligned}$$

for $\beta'_v \in [\beta_{\min}, \beta_{\max}]$ with $\beta_{\min} := \beta - \Delta^+ \leq \beta - \Delta^- =: \beta_{\max}$. Retracing our definitions, we see that $\beta_{\min} = \beta - \Delta^+ \geq 1$ holds if and only if for all $u, v \in V$ and all $I \subseteq V \setminus \{u, v\}$:

$$\beta - (C^{\uparrow}_{u,v,I} - C^{\text{uni}}_{u,v,I}) \geq 1 \iff \beta^{\downarrow} \geq \text{SINR}^{\uparrow}(u,v,I) - \text{SINR}_{\text{uni}}(u,v,I) + 1$$

where $\text{SINR}^{\uparrow}(u,v,I) := C^{\uparrow}_{u,v,I} + \beta^{\downarrow}$ is the maximum value of $\text{SINR}_{\text{adv}}(u,v,I)$ with u, v, I fixed. Hence the requirement $\beta_{\min} \geq 1$ is met iff

$$\beta^{\downarrow} \geq \max_{u,v\in V,\, I\subseteq V}\left(\text{SINR}^{\uparrow}(u,v,I) - \text{SINR}_{\text{uni}}(u,v,I) + 1\right),$$

which completes the proof. ◀

Conveniently, our construction allows us to reduce the case where the default SINR parameters in the network are *non-uniform* (e.g. nodes have different sending Power P or background noise N) to the *uniform* case without changes to the model. This can be done by transferring deviation in the connection strengths due to non-uniformity into the adversaries control. Consequently, the range of deviation $[\Delta^-, \Delta^+]$ from which the adversary may choose the deviation Δ of the true $C_{u,v,I}$ from the presumed uniform connection strength $C^{\text{uni}}_{u,v,I}$ may become large depending on the extent of non-uniformity, thereby increasing the ratio $\beta_{\max}/\beta_{\min} = (\beta - \Delta^-)/(\beta - \Delta^+)$ (intuitively speaking: we 'bought' uniformity with additional adversary influence). We saw in the previous sections that $\beta_{\max}/\beta_{\min}$ affects the run-time at most linearly.[15]

Obviously, the construction in the proof of Theorem 16 was simplified by accounting for the *largest possible* deviation of the true $C_{u,v,I}$ from the uniform connection strength $C^{\text{uni}}_{u,v,I}$ for any pair $(u, v) \in V^2, u \neq v$ and any interfering subset $I \subseteq V$. This may result in a much bigger ratio $\beta_{\max}/\beta_{\min}$ compared to the average deviation (again some intuition: we exchanged simplicity for adversary influence).

Moreover, we point out that the qualitative construction in the proof (intentionally) omits the intricate analysis of the dependence of $\beta_{\max}/\beta_{\min}$ on the SINR parameters, the network's size, design and its layout. The ratio $\beta_{\max}/\beta_{\min}$ obtained by the above construction does in fact functionally depend on these parameters and might become large. However, we argue

[14] In practice, Δ^+ and Δ^- could be determined via measurements.

[15] To minimize run-time, the choice of uniform SINR parameters P, N, α, β should minimize $\frac{\beta_{\max}}{\beta_{\min}}$.

that the size of $\beta_{\max}/\beta_{\min}$ is moderate if the local SINR parameters $P_v, N_v, \alpha(u,v), \beta_v$ are relatively homogeneous and have a narrow range of variation due to the adversary.

In order to guarantee $\beta_{\min} \geq 1$, the default sensitivity parameter β of the network devices needs to be designed large enough, so that $\beta^{\downarrow}$ does not decrease below the value given in the proof. This shows that our construction has an inherent trade-off, where increased adversary influence due to the aforementioned effects must be compensated with decreased sensitivity to signal reception (and thus also decreased transmission range).

References

1. L. Anantharamu, B. S. Chlebus, D. R. Kowalski, and M. A. Rokicki. Medium access control for adversarial channels with jamming. In *Proceedings of the 18th International Conference on Structural Information and Communication Complexity*, SIROCCO '11, 2011.
2. Baruch Awerbuch, Andréa W. Richa, and Christian Scheideler. A jamming-resistant MAC protocol for single-hop wireless networks. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 45–54. ACM, 2008. doi:10.1145/1400751.1400759.
3. Keren Censor-Hillel, Seth Gilbert, Fabian Kuhn, Nancy A. Lynch, and Calvin C. Newport. Structuring unreliable radio networks. *Distributed Computing*, 27(1):1–19, 2014. doi:10.1007/s00446-013-0198-8.
4. Sebastian Daum, Seth Gilbert, Fabian Kuhn, and Calvin C. Newport. Broadcast in the ad hoc SINR model. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2013. doi:10.1007/978-3-642-41527-2_25.
5. Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport. Secure communication over radio channels. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 105–114. ACM, 2008. doi:10.1145/1400751.1400767.
6. P. Erdös, P. Frankl, and Z. Füredi. Families of finite sets in which no set is covered by the union of r others. *Israel Journal of Mathematics*, 51(1), 1985. doi:10.1007/BF02772959.
7. Mohsen Ghaffari, Erez Kantor, Nancy A. Lynch, and Calvin C. Newport. Multi-message broadcast with abstract MAC layers and unreliable links. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 56–65. ACM, 2014. doi:10.1145/2611462.2611492.
8. Mohsen Ghaffari, Nancy A. Lynch, and Calvin C. Newport. The cost of radio network broadcast for different models of unreliable links. In Panagiota Fatourou and Gadi Taubenfeld, editors, *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 345–354. ACM, 2013. doi:10.1145/2484239.2484259.
9. Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport. Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In Alexander A. Shvartsman, editor, *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings*, volume 4305 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2006. doi:10.1007/11945529_16.
10. Olga Goussevskaia, Thomas Moscibroda, and Roger Wattenhofer. Local broadcasting in the physical interference model. In Michael Segal and Alexander Kesselman, editors, *Proceed-*

ings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing, Toronto, Canada, August 18-21, 2008, pages 35–44. ACM, 2008. doi:10.1145/1400863.1400873.

11 Magnús M. Halldórsson, Stephan Holzer, and Nancy A. Lynch. A local broadcast layer for the SINR network model. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 129–138. ACM, 2015. doi:10.1145/2767386.2767432.

12 Magnús M. Halldórsson and Pradipta Mitra. Towards tight bounds for local broadcasting. In Fabian Kuhn and Calvin C. Newport, editors, *FOMC'12, The Eighth ACM International Workshop on Foundations of Mobile Computing (part of PODC 2012), Funchal, Portugal, July 19, 2012, Proceedings*, page 2. ACM, 2012. doi:10.1145/2335470.2335472.

13 Tomasz Jurdzinski and Dariusz R. Kowalski. Distributed backbone structure for algorithms in the SINR model of wireless networks. In Marcos K. Aguilera, editor, *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, volume 7611 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2012. doi:10.1007/978-3-642-33651-5_8.

14 Tomasz Jurdzinski and Dariusz R. Kowalski. Distributed randomized broadcasting in wireless networks under the SINR model. In *Encyclopedia of Algorithms*, pages 577–580. Springer, 2016. doi:10.1007/978-1-4939-2864-4_604.

15 Tomasz Jurdzinski, Dariusz R. Kowalski, Michal Rozanski, and Grzegorz Stachowiak. On the impact of geometry on ad hoc communication in wireless networks. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 357–366. ACM, 2014. doi:10.1145/2611462.2611487.

16 Fabian Kuhn, Nancy A. Lynch, and Calvin C. Newport. The abstract MAC layer. *Distributed Computing*, 24(3-4):187–206, 2011. doi:10.1007/s00446-010-0118-0.

17 Fabian Kuhn, Nancy A. Lynch, Calvin C. Newport, Rotem Oshman, and Andréa W. Richa. Broadcasting in unreliable radio networks. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 336–345. ACM, 2010. doi:10.1145/1835698.1835779.

18 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

19 Nancy A. Lynch and Calvin Newport. A (truly) local broadcast layer for unreliable radio networks. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 109–118. ACM, 2015. doi:10.1145/2767386.2767411.

20 Dominic Meier, Yvonne Anne Pignolet, Stefan Schmid, and Roger Wattenhofer. Speed dating despite jammers. In Bhaskar Krishnamachari, Subhash Suri, Wendi Rabiner Heinzelman, and Urbashi Mitra, editors, *Distributed Computing in Sensor Systems, 5th IEEE International Conference, DCOSS 2009, Marina del Rey, CA, USA, June 8-10, 2009. Proceedings*, volume 5516 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2009. doi:10.1007/978-3-642-02085-8_1.

21 T. Moscibroda, R. Wattenhofer, and Y. Weber. Protocol design beyond graph-based models. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets-V)*, 2006.

22 Calvin C. Newport, David Kotz, Yougu Yuan, Robert S. Gray, Jason Liu, and Chip Elliott. Experimental evaluation of wireless simulation assumptions. *Simulation*, 83(9):643–661, 2007. doi:10.1177/0037549707085632.

23 Adrian Ogierman, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. Competitive MAC under adversarial SINR. In *2014 IEEE Conference on Computer Com-*

munications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014, pages 2751–2759. IEEE, 2014. doi:10.1109/INFOCOM.2014.6848224.

24 A. Richa, Ch. Scheideler, S. Schmid, and J. Zhang. A jamming-resistant MAC protocol for multi-hop wireless networks. In *Proceedings of the 24th International Conference on Distributed Computing*, DISC '10. Springer-Verlag, 2010.

25 D. Yu, Q. Hua, Y. Wang, and F. C. M. Lau. An O(log n) distributed approximation algorithm for local broadcasting in unstructured wireless networks. In *IEEE 8th International Conference on Distributed Computing in Sensor Systems*, ICDCS '12. IEEE, 2012. doi:10.1109/dcoss.2012.39.

26 Dongxiao Yu, Qiang-Sheng Hua, Yuexuan Wang, Haisheng Tan, and Francis C. M. Lau. Distributed multiple-message broadcast in wireless ad hoc networks under the SINR model. *Theor. Comput. Sci.*, 610:182–191, 2016. doi:10.1016/j.tcs.2014.06.043.

A Δ-Cover-Free Families

Proof. (Lemma 8). Let $\mathbb{F}_q$ be the finite field with q elements. Let $\mathcal{P}_d = \{a_0+a_1x+\ldots+a_dx^d \mid a_i \in \mathbb{F}_q\}$ be the set of polynomials with coefficients in $\mathbb{F}_q$ and degree at most d. For a polynomial $f \in \mathcal{P}_d$ let $G_f := \{(x, f(x)) \mid x \in \mathbb{F}_q\} \subseteq \mathbb{F}_q^2$ be the graph of f over $\mathbb{F}_q$. Consider the family $\mathcal{F} := \{G_f \mid f \in \mathbb{F}_{q^d}\} \subseteq 2^{\mathbb{F}_q^2}$ of graphs of polynomials from $\mathcal{P}_d$ over $\mathbb{F}_q$.

Graphs of degree d or less intersect at most d times and since $d < q$, all graphs G_f of polynomials $f \in \mathcal{P}_d$ are distinct, hence $|\mathcal{F}| = |\mathcal{P}_d| = q^{d+1}$. For the same reason we have $G_1 \cap G_2 \leq d$ for graphs $G_1, G_2 \in \mathcal{F}$, hence $\lfloor (q-1)/d \rfloor$ graphs are cover-free. ◀

Proof. (Lemma 9). We choose the smallest prime power $q \geq \Delta \log k + 1$. Considering powers of 2 we obviously have $\Delta \log k + 1 \leq q \leq 3\Delta \log k$. Moreover, we choose $d := \log k$. Then we have $\lfloor (q-1)/d \rfloor \geq \lfloor (\Delta \log k)/\log k \rfloor = \Delta$, and due to Lemma 8, there is a Δ-cover-free family $\mathcal{F}$ of size $|\mathcal{F}| = q^{d+1} \geq (\Delta \log k + 1)^{\log k} \geq k$ of subsets of $[q^2]$. ◀

B Neighborhood Dissemination Analysis

Proof. (Lemma 12) The case $|S_\phi| \leq 1$ is covered, since $d_\phi = \infty \geq 2^{\phi-1} d_{\min}$. In case $|S_\phi| \geq 2$ we prove the lemma by induction on ϕ. Obviously, we have $d_1 \geq d_{\min} = 2^0 d_{\min}$. Presume that $d_\phi \geq 2^{\phi-1} d_{\min}$ or $d_\phi > r_e$. If $d_\phi > r_e$ is true, we have $d_{\phi+1} \geq d_\phi > r_e$ and are done. Otherwise, we have $d_\phi \geq 2^{\phi-1} d_{\min}$. In Lemma 10 we showed that $S_{\phi+1}$ is a DIS of S_ϕ, thus independent w.r.t. $E^\mu_{\text{safe}}[S_\phi]$. Therefore no two nodes from $S_{\phi+1}$ are connected via a μ-safe edge in $H^\mu[S_\phi]$.

For a contradiction, assume $d_{\phi+1} < 2d_\phi$ and $d_{\phi+1} \leq r_e$. There are two nodes $u, v \in S_{\phi+1}$ with $d(u, v) = d_{\phi+1}$. Let $\mu \in (0, p)$ be the constant from Lemma 4, such that $E^\mu_{\text{safe}}[S_\phi]$ contains all edges not longer than than $\min(2d_\phi, r_e)$. Thus, u, v would be connected via a μ-safe edge in $H^\mu[S_\phi]$, a contradiction to the independence of the DIS $S_{\phi+1}$. Consequently, $d_{\phi+1} \geq 2d_\phi \geq 2^\phi d_{\min}$ (second inequality is due to the induction hypothesis) or $d_{\phi+1} > r_e$. ◀

Proof. (Lemma 11). If $X_A^p > \frac{2^\alpha}{\beta_{\min}}$, then u cannot receive a message from w, because

$$\text{SINR}(w, u, I) \leq \frac{P/d(u,w)^\alpha}{I_A^p(u)} \leq \frac{P/d(u,w)^\alpha}{X_A^p P/2^\alpha d(u,w)^\alpha} < \beta_{\min}.$$

However, since $(w, u) \in E^\mu[S_\phi]$, we have $\text{SINR}(w, u, I) \geq \beta_{\min}$ with probability at least $(1-\varepsilon)\mu$ and therefore $\mathbb{P}(X_A^p \leq \frac{2^\alpha}{\beta_{\min}}) \geq (1-\varepsilon)\mu$. Now assume nodes send with probability p/Q instead of p. We simulate this with a two step random experiment. First, we randomly

determine a set $C \subseteq A$ of candidate nodes, whereas the probability that $x \in A$ becomes a candidate is p. Second, the probability that a candidate $x \in C$ actually sends is $1/Q$. Using the law of total probability and then Markov's inequality we obtain

$$\begin{aligned}\mathbb{P}\left(X_A^{p/Q} \leq \frac{2^{\alpha+1}}{Q\beta_{\min}}\right) &\geq \mathbb{P}\left(|C| \leq \frac{2^\alpha}{\beta_{\min}}\right) \cdot \mathbb{P}\left(X_C^{1/Q} \leq \frac{2^{\alpha+1}}{Q\beta_{\min}} \;\middle|\; |C| \leq \frac{2^\alpha}{\beta_{\min}}\right) \\ &\geq \mathbb{P}\left(X_A^p \leq \frac{2^\alpha}{\beta_{\min}}\right) \cdot \mathbb{P}\left(X_C^{1/Q} \leq 2\mathbb{E}(X_C^{1/Q})\right) \geq \frac{(1-\varepsilon)\mu}{2}.\end{aligned}$$

Therefore, with probability $\frac{(1-\varepsilon)\mu}{2}$ the interference $I_A^{p/Q}(v)$ is bounded by

$$I_A^{p/Q}(v) = \sum_{x \in A \text{ sends}} \frac{P}{d(x,v)^\alpha} \leq \frac{X_A^{p/Q} P}{d(u,v)^\alpha} = \frac{2^{\alpha+1} P}{Q\beta_{\min} d(u,v)^\alpha}.$$

Next, we limit the maximum interference $I_{\bar{A}}^p(u)$ at u stemming from the set of distant nodes $\bar{A}$. Again, we utilize that $(w,u) \in E^\mu[S_\phi]$.

$$\begin{aligned}(1-\varepsilon)\mu \leq \mathbb{P}\left(SINR(w,u,I) \geq \beta_{\min}\right) &= \mathbb{P}\left(\frac{P/d(u,w)^\alpha}{N + I_V^p(u)} \geq \beta_{\min}\right) \\ &\leq \mathbb{P}\left(\frac{P}{I_{\bar{A}}^p(u) d(u,w)^\alpha} \geq \beta_{\min}\right) = \mathbb{P}\left(I_{\bar{A}}^p(u) \leq \frac{P}{\beta_{\min} d(u,w)^\alpha}\right).\end{aligned} \tag{3}$$

Using the specific Chernoff bound from [4] given in Appendix C Lemma 18, we obtain

$$\mathbb{P}\left(I_{\bar{A}}^p(u) \leq \frac{\mathbb{E}(I_{\bar{A}}^p(u))}{2}\right) \leq \mathbb{P}\left(I_{\bar{A}}^p(u) \leq \frac{pI_{\bar{A}}^{\max}(u)}{2}\right) \leq \exp\left(-\frac{p2^\alpha d(u,w)^\alpha}{8P} \cdot I_{\bar{A}}^{\max}(u)\right). \tag{4}$$

We show $I_{\bar{A}}^{\max}(u) \leq \frac{\eta \cdot P}{p\beta_{\min} d(u,w)^\alpha}$ with $\eta \coloneqq \max\left\{2, \frac{8\beta_{\min}}{2^\alpha} \cdot \ln \frac{1}{(1-\varepsilon)\mu}\right\} \in \Theta(\beta_{\min})$. Assume that $I_{\bar{A}}^{\max}(u) > \frac{\eta \cdot P}{p\beta_{\min} d(u,w)^\alpha}$. Then we obtain a contradiction to Equation 3 as follows

$$\begin{aligned}\mathbb{P}\left(I_{\bar{A}}^p(u) \leq \frac{P}{\beta_{\min} d(u,w)^\alpha}\right) &\overset{\eta \geq 2}{\leq} \mathbb{P}\left(I_{\bar{A}}^p(u) \leq \frac{\eta \cdot P}{2\beta_{\min} d(u,w)^\alpha}\right) \leq \mathbb{P}\left(I_{\bar{A}}^p(u) \leq \frac{pI_{\bar{A}}^{\max}(u)}{2}\right) \\ &\overset{(4)}{\leq} \exp\left(-\frac{p2^\alpha d(u,w)^\alpha}{8P} I_{\bar{A}}^{\max}(u)\right) < \exp\left(-\frac{2^\alpha \eta}{8\beta_{\min}}\right) \leq (1-\varepsilon)\mu.\end{aligned}$$

The upper bound of $I_{\bar{A}}^{\max}(u)$ can be used to bound $I_{\bar{A}}^{\max}(v)$. For any node $x \in S_\phi$ we have $d(u,x) \leq d(u,v) + d(v,x) \leq 2d(v,x)$, since u is closest to v among the nodes in S_ϕ. Therefore, we obtain

$$I_{\bar{A}}(v) = \sum_{x \in \bar{A}} \frac{P}{d(v,x)^\alpha} \leq \sum_{x \in \bar{A}} \frac{2^\alpha P}{d(u,x)^\alpha} = 2^\alpha I_{\bar{A}}(u).$$

Markov's inequality yields

$$\mathbb{P}\left(I_{\bar{A}}^{p/Q}(v) \leq 2\mathbb{E}\left(I_{\bar{A}}^{p/Q}(v)\right)\right) \leq \mathbb{P}\left(I_{\bar{A}}^{p/Q}(v) \leq \frac{2p}{Q} I_{\bar{A}}^{\max}(v)\right) \geq \frac{1}{2}.$$

Thus, with probability at least $\frac{1}{2}$ we have $I_{\bar{A}}^{p/Q}(v) \leq \frac{2p}{Q} I_{\bar{A}}^{\max}(v) \leq \frac{2^{\alpha+1}p}{Q} I_{\bar{A}}^{\max}(u) \leq \frac{2^{\alpha+1}\eta P}{Q\beta_{\min} d(u,w)^\alpha}$. Finally, the probability that u sends and v listens is $\frac{p}{Q}(1-\frac{p}{Q}) \leq \frac{p}{2Q}$. Events (i), (ii) and (iii) are independent, hence the probability that all of them to occur simultaneously is $\frac{(1-\varepsilon)\mu p}{8Q}$. ◀

Proof. (Lemma 13). Assume that events (i), (ii) and (iii) from Lemma 12 occur. We use the bounds on the interference to show that a safe transmission from u to v takes place. This is the case if $\mathrm{SINR}(u,v,I) \geq \beta_{\max}$, i.e., $\beta_{\max} d(u,v)^\alpha (N + I_A^{p/Q}(v) + I_{\bar{A}}^{p/Q}(v)) \leq P$.

$$
\begin{aligned}
&\beta_{\max} d(u,v)^\alpha \left(N + I_A^{p/Q}(v) + I_{\bar{A}}^{p/Q}(v)\right) \\
&\quad \leq \beta_{\max}\left(d(u,v)^\alpha N + \frac{2^{\alpha+1}P}{Q\beta_{\min}} + \frac{2^{\alpha+1}\eta P d(u,v)^\alpha}{Q\beta_{\min} d(u,w)^\alpha}\right) && \text{Lemma 12 (i), (ii)} \\
&\quad \leq \beta_{\max}\left((1+\frac{\rho}{2})^\alpha r_e^\alpha N + \frac{2^{\alpha+1}P}{\hat{Q}\beta_{\min}} + \frac{2^{\alpha+1}\eta P}{\gamma^\alpha \beta_{\min}}\right) && d(u,v) \leq (1+\tfrac{\rho}{2})r_e,\ d(u,v)^\alpha \leq \tfrac{Q d(u,w)^\alpha}{\gamma^\alpha} \\
&\quad \leq \beta_{\max}\left((1-\frac{\alpha\rho}{2(1+\rho)^\alpha})(1+\rho)^\alpha r_e^\alpha N + \frac{2^{\alpha+1}P}{\hat{Q}\beta_{\min}} + \frac{2^{\alpha+1}\eta P}{\gamma^\alpha \beta_{\min}}\right) && (1+\tfrac{\rho}{2})^\alpha \overset{\text{Lemma 19}}{\leq} (1+\rho)^\alpha - \tfrac{\alpha\rho}{2} \\
&\quad \leq P\left(1-\frac{\alpha\rho}{2(1+\rho)^\alpha}\right) + \beta_{\max}\left(\frac{2^{\alpha+1}P}{\hat{Q}\beta_{\min}} + \frac{2^{\alpha+1}\eta P}{\gamma^\alpha\beta_{\min}}\right) && P = \beta_{\max}(1+\rho)^\alpha r_e^\alpha N \\
&\quad = P + P\left(\frac{\beta_{\max}2^{\alpha+1}}{\beta_{\min}\hat{Q}} + \frac{\beta_{\max}2^{\alpha+1}\eta}{\beta_{\min}\gamma^\alpha} - \frac{\alpha\rho}{2(1+\rho)^\alpha}\right) \leq P && \hat{Q} \in \mathcal{O}\left(\tfrac{\beta_{\max}}{\beta_{\min}}2^\alpha\right), \gamma \in \mathcal{O}\left((\tfrac{\beta_{\max}}{\beta_{\min}})^{1/\alpha}\right).
\end{aligned}
$$

In the last step, $\hat{Q} \in \mathcal{O}(\frac{\beta_{\max}}{\beta_{\min}}2^\alpha), \gamma \in \mathcal{O}((\frac{\beta_{\max}}{\beta_{\min}})^{1/\alpha})$ are chosen sufficiently large, such that the term in the bracket becomes negative. Due to event (iii) u sends and v listens, thus v receives $\mathcal{M}$ from u. Events (i), (ii) and (iii) occur with probability $\frac{(1-\epsilon)\mu p}{8Q} \in \Theta(1/Q)$. ◀

Proof. (Lemma 14) Note that we assume $\log R \geq 1$ (otherwise the network is 'sparse' and broadcast is easy, see Lemma 1). We proof the claim inductively. Active node $u_1 \in S_1$ is a neighbor of $v \in N(S_1)$ in the communication graph G_C, thus $d(u_1,v) \leq r_e$ by definition. Presume that during phase ϕ either $d(u_\phi, v) \leq r_e(1 + \frac{\rho(\phi-1)}{2\log R})$ or v has already received $\mathcal{M}$. We show that the same is true for phase $\phi+1$. If v has already received $\mathcal{M}$ we are done. Therefore, we concentrate on the case that $d(u_\phi, v)$ is bound from above.

If there is no active node $w_\phi \in S_\phi$ for which an edge $(w_\phi, u_\phi) \in E^\mu[S_\phi]$ exists, then $u_{\phi+1}$ is not dominated by a node in S_ϕ and therefore $u_{\phi+1} = u_\phi$. With to the induction hypothesis, we obtain $d(u_{\phi+1}, v) \leq r_e(1+\frac{\rho(\phi-1)}{2\log R}) \leq r_e(1+\frac{\rho\phi}{2\log R})$. Otherwise, let w_ϕ to be the farthest node from u_ϕ with an edge $(w_\phi, u_\phi) \in E^\mu[S_\phi]$.

If $d(u_\phi, w_\phi) \geq \gamma Q^{-1/\alpha} d(u_\phi, v)$, then Lemma 13 applies and we have a guaranteed probability of $\Theta(1/Q)$ that v receives $\mathcal{M}$, in case active nodes send with probability p/Q. Since we do exactly that for $\Theta(Q \log n)$ rounds during phase ϕ of algorithm RobustDissemination, node v receives $\mathcal{M}$ w.h.p. (we adjust the constant in $\Theta(Q \log n)$ accordingly).

Now consider the case $d(u_\phi, w_\phi) < \gamma Q^{-1/\alpha} d(u_\phi, v)$. Since $u_{\phi+1} \neq u_\phi$, node u_ϕ is dominated by some other node $x \in S_{\phi+1}$, which is at distance at most $d(u_\phi, w_\phi)$ from u_ϕ. Thus, the nearest neighbor $u_{\phi+1} \in S_{\phi+1}$ of v is at distance at most $d(v, u_{\phi+1}) \leq d(v,x) \leq d(v,u_\phi) + d(u_\phi, x) \leq d(v,u_\phi) + d(u_\phi, w_\phi)$. We obtain

$$
\begin{aligned}
d(v, u_{\phi+1}) &\leq \left(1 + \frac{\gamma}{Q^{1/\alpha}}\right) d(v, u_\phi) \leq r_e \left(1 + \frac{\gamma}{Q^{1/\alpha}}\right)\left(1 + \frac{\rho(\phi-1)}{2\log R}\right) \\
&\overset{(*)}{\leq} r_e\left(1 + \frac{\rho(\phi-1)}{2\log R} + \frac{\gamma(1+\rho/2)}{Q^{1/\alpha}}\right) \leq r_e\left(1 + \frac{\rho\phi}{2\log R}\right) && (*) : \phi - 1 \leq \log R
\end{aligned}
$$

The last step holds for a sufficiently large $Q \in \Theta(\frac{\beta_{\max}}{\beta_{\min}}(\log R)^\alpha), Q \geq \hat{Q}$. ◀

Proof. (Lemma 15) Since S_ϕ is sparse the premise of Lemma 1 is fulfilled and v receives $\mathcal{M}$ from u with constant probability $\mu' = \frac{p}{Q}(1-\frac{p}{Q})^{k_0}$ and $k_0 \in \Theta(1)$ (nodes send with probability p/Q in each phase of Algorithm 1). Therefore, v receives $\mathcal{M}$ from u w.h.p. after sending

with probability p/Q for $\mathcal{O}(\frac{\log n}{\mu'})$ rounds.[16] Define the constant probability $\mu := p(1-p)^{k_0}$. We find

$$\mu' = \frac{p}{Q}(1-\frac{p}{Q})^{k_0} \geq \frac{p}{Q}(1-p)^{k_0} = \frac{\mu}{Q}.$$

Hence $(\log n)/\mu' \leq (Q\log n)/\mu$. Therefore, v also receives $\mathcal{M}$ w.h.p. if u sends with probability p/Q for $\mathcal{O}(\frac{Q\log n}{\mu})$ rounds (i.e. longer). Since u sends for $\Theta(Q\log n)$ rounds in the inner loop of Algorithm 1 (with an appropriately chosen constant factor), we have proven the claim. ◀

C Inequalities

For the sake of completeness, the following lemmas give the specific bounds which we use in the proofs of Lemma 1, Lemma 12 and Lemma 13. We refer to [4] for the proof of Lemma 18.

▶ **Lemma 17.** *Let $\delta \geq 1$ and $k \in \mathbb{N}$. Then $k^\delta - (k-1)^\delta \leq \delta k^{\delta-1}$.*

Proof. Let $f(k) := k^\delta$. Then $f'(k) := \frac{df(k)}{dk} = \delta k^{\delta-1}$ is monotonous increasing since $\delta \geq 1$. The mean value theorem states that there is a $\xi \in (k-1,k)$, such that $f(k) - f(k-1) = f'(\xi)(k-(k-1)) = f'(\xi) \leq f'(k)$. ◀

▶ **Lemma 18** (cf. [4])**.** *Let $X_1,\dots,X_k$ be independent random variables with $\mathbb{P}(X_i = a_i) = p$ and $\mathbb{P}(X_i = 0) = 1-p$ for $a_i > 0$ and $p \in (0,1)$. Let $A := \sum_{i=1}^k a_i$ and $\hat{a} := \max_{i\in[k]} a_i$. Further let $X := \sum_{i=1}^k X_i$ and $\lambda := \mathbb{E}(X) = pA$. For any $\delta > 0$ it holds that*

$$\mathbb{P}(X \leq (1-\delta)\lambda) \leq \exp(-\tfrac{\delta^2\lambda}{2\hat{a}}).$$

▶ **Lemma 19.** *Let $\rho \geq 0$ and $\alpha \geq 2$. Then $(1+\rho)^\alpha \leq (1+\frac{\rho}{2})^\alpha + \frac{\alpha\rho}{2}$.*

Proof. Let $f(x) := x^\alpha - (\frac{x+1}{2})^\alpha - \alpha\frac{x-1}{2}$. We proof the claim by showing $f(1+\rho) \geq 0$. First we note that $f(1) = 0$ fulfills the claim for $\rho = 0$. The claim is proved for $\rho \geq 0$ if we can show $f'(x) := \frac{df(x)}{dx} \geq 0$ for $x \geq 1$. We have $x \geq \frac{x+1}{2}$ (since $x \geq 1$). This is equivalent to $x^{\alpha-1} \geq \left(\frac{x+1}{2}\right)^{\alpha-1}$ (since $\alpha \geq 2$). Hence $x^{\alpha-1} - \frac{1}{2}\left(\frac{x+1}{2}\right)^{\alpha-1} \geq x^{\alpha-1} - \frac{1}{2}x^{\alpha-1} \geq \frac{1}{2}$ (since $x \geq 1, \alpha \geq 2$). Finally, we obtain that $f'(x) = \alpha\big(\underbrace{x^{\alpha-1} - \frac{1}{2}\left(\frac{x+1}{2}\right)^{\alpha-1}}_{\geq 1/2} - \frac{1}{2}\big) \geq 0$. ◀

[16] The math is similar to what we did it in the proof of Lemma 5.

Deterministic Subgraph Detection in Broadcast CONGEST*

Janne H. Korhonen[1] and Joel Rybicki[2]

1 **Aalto University, Finland**
`janne.h.korhonen@aalto.fi`
2 **University of Helsinki, Finland**
`joel.rybicki@helsinki.fi`

Abstract

We present simple deterministic algorithms for subgraph finding and enumeration in the broadcast CONGEST model of distributed computation:

- For any constant k, detecting k-paths and trees on k nodes can be done in $O(1)$ rounds.
- For any constant k, detecting k-cycles and pseudotrees on k nodes can be done in $O(n)$ rounds.
- On d-degenerate graphs, cliques and 4-cycles can be enumerated in $O(d + \log n)$ rounds, and 5-cycles in $O(d^2 + \log n)$ rounds.

In many cases, these bounds are tight up to logarithmic factors. Moreover, we show that the algorithms for d-degenerate graphs can be improved to $O(d/\log n)$ and $O(d^2/\log n)$, respectively, in the supported CONGEST model, which can be seen as an intermediate model between CONGEST and the congested clique.

1998 ACM Subject Classification C.2.4 Distributed Systems, G.2.2 Graph Theory

Keywords and phrases distributed computing, subgraph detection, CONGEST model, lower bounds

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.4

1 Introduction

Subgraph detection is a fundamental problem in algorithmics: given a fixed target graph H and an input graph G, the task is to decide whether G contains a subgraph isomorphic to H. This problem has been extensively studied in centralised algorithmics, and it is known that depending on the graph H, this problem can have very different complexities – for example, if H is a path, the problem can be solved in linear (in the number of nodes in G) time [1], but for cliques this is thought not to be the case [12].

In this work, we study this problem in the CONGEST model of distributed computation. So far, most work has either focused on (a) lower bounds for subgraph detection [13], (b) randomised upper bounds [21, 17], or (c) property testing of H-freeness [20, 8, 17]. By contrast, we focus on deterministic upper bounds, showing that simple techniques – indeed, often techniques well-known in non-distributed algorithmics – result in essentially optimal algorithms. Moreover, all our results work also in the weaker *broadcast* CONGEST model, where nodes send the same message to all neighbours in each communication round.

* This work was supported in part by the Academy of Finland, Grants 285721 and 1273253.

Results: subgraph detection on general graphs. First, we give simple constant-time detection algorithms for the case where H is either a path or a tree. These algorithms are essentially translations of known centralised *fixed-parameter* algorithms, based on *representative families* [23, 18, 24]. Specifically, we show that in the broadcast CONGEST model,

- k-paths can be detected in $O(k2^k)$ rounds, and
- any tree on k nodes can be detected in $O(k2^k)$ rounds[1].

Using these algorithms as building blocks, we then give linear-time detection algorithms for cycles and pseudotrees:

- k-cycles can be detected in $O(k2^k n)$ rounds, and
- any pseudotree (a graph with exactly one cycle) on k nodes can be detected in $O(k2^k n)$ rounds.

All algorithms can be implemented so that any node that detects the existence of a copy of H can also output full information about a single copy of H.

For odd $k \geq 5$, Drucker et al. [13] have proven a lower bound of $\Omega(n/\log n)$ rounds for detecting k-cycles, meaning that our cycle detection algorithm is optimal up to a logarithmic factor for a constant odd k. For 4-cycles, Drucker et al. [13] gave a lower bound of $\Omega(n^{1/2}/\log n)$ rounds, and a weaker lower bound for longer even cycles (see Section 2); we prove a simple extension of the Drucker et al. [13] lower bound by showing that for all even $k \geq 6$, detecting k-cycles requires $\Omega(n^{1/2}/\log n)$ rounds in the CONGEST model.

Results: subgraph enumeration on sparse graphs. Second, we study subgraph detection and enumeration on sparse input graphs. Specifically, we study a setting where the input graph has small *degeneracy*; in distributed and parallel computing, this setting has been studied in the context of e.g. symmetry breaking in the LOCAL model [2, 4] and enumerating triangles and 4-cliques of planar graphs in the PRAM model [10]. Denoting by d the degeneracy of the input graph, we show that in the broadcast CONGEST model,

- k-cliques can be enumerated in $O(d + \log n)$ rounds for any k,
- 3-cycles and 4-cycles can be enumerated in $O(d + \log n)$ rounds, and
- 5-cycles can be enumerated in $O(d^2 + \log n)$ rounds,

where enumeration means that every copy of the target subgraph H is output by some node in the network.

This is as far as we can push this framework; we show that already detecting k-cycles for $k \geq 6$ requires $\Omega(n^{1/2}/\log n)$ rounds on graphs of degeneracy 2. Moreover, a careful examination of the Drucker et al. [13] lower bound constructions show that detecting 4-cycles and 5-cycles requires $\Omega(d/\log n)$ rounds.

Finally, we discuss how the results on detecting subgraphs in sparse graphs can be translated into the *supported* CONGEST *model* proposed by Schmid and Suomela [29]. In this model, we can close many of the logarithmic gaps between the upper and lower bounds that remain in the CONGEST model.

---

[1] We note that the constant-round algorithms for path and tree detection were also independently discovered by Fraigniaud et al. [19]; however, we give more detailed analysis in terms of the factors dependent on k. In fact, very similar algorithms were independently discovered in *four* separate works almost concurrently, including this one [22] and the papers of Fraigniaud et al. [19], Even et al. [16] and Fischer et al. [17]. The last three appear as a joint paper in DISC 2017 [15]. While our work is otherwise independent of the others, our pseudotree detection algorithm is motivated by the pseudotree property testing results of Fraigniaud et al. [19].

2 Related work

Subgraph detection upper bounds. For deterministic subgraph detection in the CONGEST model, the only prior works we are aware of are the $O(n^{1/2})$ round algorithm for 4-cycle detection by Drucker et al. [13], and the independent discovery of the constant-round path and tree detection algorithms [19, 15]. In the *congested clique* model, deterministic subgraph detection algorithms were given by Dolev et al. [11] and Censor-Hillel et al. [9]; the latter is in particular noteworthy from our perspective, as it used *colour-coding* techniques from centralised fixed-parameter algorithmics to obtain fast cycle detection algorithms for cycles of arbitrary length.

Randomised subgraph detection and listing in the CONGEST model has been recently receiving attention from multiple authors. Izumi and Le Gall [21] gave an $\tilde{O}(n^{2/3})$ round algorithm for detecting triangles, and a $O(n^{3/4}\log n)$ round algorithm for enumerating triangles. Independent of our work, Fischer et al. [17] gave a colour-coding algorithm that can detect any constant-size tree on k nodes with constant probability in $O(k^k)$ rounds.

Subgraph detection lower bounds. As noted before, Drucker et al. [13] have studied lower bounds for cycle detection in the CONGEST model. Specifically, their lower bound for k-cycle detection is $\Omega(\mathrm{ex}(n, C_k)/n \log n)$ rounds, where $\mathrm{ex}(n, C_k)$ is the *Turán number* for cycles, that is, the maximum number of edges in a k-cycle-free graph with n nodes. This lower bound is $\Omega(n/\log n)$ for odd cycles, and $\Omega(n^{1/2}/\log n)$ for 4-cycles. However, for longer even cycles, this can give at most $\Omega(n^{1+2/k}/\log n)$ due to known bounds for Turán numbers, and matching bounds on Turán numbers are only known for $k = 6$ and $k = 8$; see e.g. Pikhurko [28] and references therein.

To our knowledge, no other subgraph detection lower bounds are known in the CONGEST model. In particular, proving lower bounds for triangle detection seems to be a particularly difficult challenge. However, for the broadcast congested clique model, Drucker et al. [13] give an $\Omega(n/e^{\sqrt{\log n}}\log n)$ round lower bound, which also applies to the broadcast CONGEST model. Moreover, for triangle enumeration lower bounds are known, also in the stronger congested clique model [21, 26].

Property testing for H-freeness. Property testing of H-freeness in the CONGEST model is another question that has received a lot of attention lately. In this setting, an algorithm has to correctly decide with probability at least 2/3 if the input graph is (a) H-free – that is, does not contain a subgraph isomorphic to H – or (b) ε-away from being H-free; in the intermediate case, the algorithm can perform arbitrarily. See e.g. Censor-Hillel et al. [8] for complete definitions.

Property testing algorithms for triangle-freeness were given by Censor-Hillel et al. [8]; for H-freeness for graphs on 4 nodes by Fraigniaud et al. [19] and Even et al. [16]; and for H-freeness for most graphs on 5 nodes by Fischer et al. [17] Moreover, property testing algorithms for tree and cycle freeness – using techniques of fixed-parameter algorithmics flavour – have been discovered recently [16, 17, 19, 15].

3 Preliminaries

Set notation. Let $\mathbb{N} = \{0, 1, \ldots\}$ be the set of non-negative integers. For integer $n \geq 1$, we use the notation $[n] = \{1, 2, \ldots, n\}$. For any set A, we use $2^A = \{B : B \subseteq A\}$ to denote the power set of A.

Graphs. A graph is a pair $G = (V, E)$, where V is the set of nodes and $E \subseteq 2^V$ is the set of edges. We use $n = |V|$ to denote the number of nodes in the graph. An *orientation* σ of graph G is a labelling of the edges that assigns a direction $\sigma(\{u, v\}) \in \{u \to v, u \leftarrow v\}$ to every edge $\{u, v\} \in E$.

The open neighbourhood of a node $v \in V$ is the set $N(v) = \{u \in V : \{u, v\} \in E\}$ and the closed neighbourhood is the set $N^+(v) = N(v) \cup \{v\}$. The degree of a node $v \in V$ is $\deg(v) = |N(v)|$. The neighbours of $v \in V$ with incoming and outgoing edges under an orientation σ are denoted by $N_{\text{in}}(v) = \{u \in N(v) : \sigma(\{u, v\}) = u \to v\}$ and $N_{\text{out}}(v) = N(v) \setminus N_{\text{in}}(v)$. The indegree of $v \in V$ under an orientation σ is $\text{indeg}(v) = |N_{\text{in}}(v)|$ and the outdegree is given by $\text{outdeg}(v) = |N_{\text{out}}(v)|$.

For a graph G, a subgraph $G' \subseteq G$ of G is a graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E \cap 2^{V'}$. The subgraph induced by the node set $A \subseteq V$ is $G[A] = (A, E')$, where $E' = \{e \in E : e \subseteq A\}$. Similarly, the subgraph induced by an edge set $F \subseteq E$ is $G[F] = (V', F)$, where $V' = \{u \in e : e \in F\}$. A graph G is *d-degenerate* if every subgraph $G' \subseteq G$ contains a node with degree at most d. Every graph is trivially $(n-1)$-degenerate. We note that degeneracy is within a constant factor of an another widely-used graph parameter *arboricity*, that is, any graph with arboricity a has degeneracy $a \leq d \leq 2a - 1$; see e.g. Barenboim and Elkin [3].

CONGEST and broadcast CONGEST. The CONGEST model is a variant of classical LOCAL model of distributed computation with additional constraints on communication bandwidth [27]. The distributed system is represented as a network $G = (V, E)$, where each node $v \in V$ executes the same algorithm in synchronous rounds, and the nodes collaborate to solve a graph problem with input G. Each round, all nodes

1. perform an unlimited amount of local computation,
2. send a possibly different $O(\log n)$-bit message to each of their neighbours, and
3. receive the messages sent to them.

The time measure is the number of synchronous rounds required. Each node is assumed to have a unique identifiers from the set $\{0, \ldots, \text{poly}(n)\}$.

The broadcast CONGEST is a weaker version of CONGEST, with the additional constraint that all nodes have to send the same message to each of their neighbours.

4 Finding paths, cycles and trees

Representative families. The general cycle and path detection algorithms are based on *representative families* [24, 14]. For a basic intuition, consider a setting where we have a collection of objects (in this case, sets) of size p, and we want to extend them by adding at most q more elements. Representative families allow us to 'compress' our collection of objects so that if a member of the original collection could be extended by specific q elements, then the compressed collection also contains a member that can be extended by the same elements. Indeed, while we only use the set family version of this theory, it can be extended to *matroids* [18].

▶ **Definition 1.** Let $\mathcal{A} \subseteq 2^{[n]}$. We say that $\widehat{\mathcal{A}} \subseteq \mathcal{A}$ is *q-representative* for $\mathcal{A}$ if for each $B \subseteq [n]$ with $|B| \leq q$, there is a set $A \in \mathcal{A}$ with $A \cap B = \emptyset$ if and only if there is $\widehat{A} \in \widehat{\mathcal{A}}$ with $\widehat{A} \cap B = \emptyset$.

▶ **Theorem 2** ([14]). *Let $\mathcal{A} \subseteq 2^{[n]}$ consist of sets of size at most p. Then there is a q-representative $\widehat{\mathcal{A}} \subseteq \mathcal{A}$ with $|\widehat{\mathcal{A}}| \leq \binom{p+q}{p}$.*

For our purposes it is sufficient to know that small representative families exist; however, slightly worse bounds with corresponding efficient algorithms are known [23, 18, 14].

Finally, we will use the following simple fact about representative families:

▶ **Lemma 3.** *Let $\mathcal{C} \subseteq \mathcal{B} \subseteq \mathcal{A} \subseteq 2^{[n]}$. If $\mathcal{B}$ is q-representative for $\mathcal{A}$ and $\mathcal{C}$ is q-representative for $\mathcal{B}$, then $\mathcal{C}$ is q-representative for $\mathcal{A}$.*

Together Theorem 2 and Lemma 3 imply that any inclusion-minimal q-representative family for a family of sets of size at most p has size at most $\binom{p+q}{p}$.

Finding paths. We start by giving a simple path detection algorithm for the CONGEST model, using a well-known technique from fixed-parameter algorithmics [24]. Intuitively, the idea is to start with the trivial path detection algorithm that iteratively propagates full information about paths of length at most $\ell = 1, 2, \ldots, k$ in k phases. This will be quite slow if there are lots of such paths; however, at each step, we construct a representative family for the current set of paths to ensure that we only forward the essential ones.

Let k be fixed, and define for $v \in V$ and $\ell \in \{1, 2, \ldots, k\}$ the set family

$$\mathcal{P}_{v,\ell} = \left\{U \subseteq V : \text{there is an } \ell\text{-path with node set } U \text{ that ends at } v\right\}.$$

By definition, we have that

$$\mathcal{P}_{v,\ell} = \bigcup_{u \in N(v)} \left\{\{v\} \cup U : U \in \mathcal{P}_{u,\ell-1} \text{ and } v \notin U\right\}.$$

The basic idea of the algorithm is that instead of explicitly constructing the families $\mathcal{P}_{v,1}, \mathcal{P}_{v,2} \ldots, \mathcal{P}_{v,k}$, we iteratively construct families $\widehat{\mathcal{P}}_{v,1}, \widehat{\mathcal{P}}_{v,2} \ldots, \widehat{\mathcal{P}}_{v,k}$, where $\widehat{\mathcal{P}}_{v,\ell}$ is a minimal $(k-\ell)$-representative family for $\mathcal{P}_{v,\ell}$.

Specifically, the algorithm proceeds as follows:

1. For $\ell = 1$, we have that $\mathcal{P}_{v,1} = \{\{u, v\} : u \in N(v)\}$. Each node v can obtain full information about $N(v)$ in single round, and then compute $\widehat{\mathcal{P}}_{v,1}$ locally.
2. For $\ell \geq 2$, after the families $\widehat{\mathcal{P}}_{v,\ell-1}$ have been constructed, each node $v \in V$ broadcasts the family $\widehat{\mathcal{P}}_{v,\ell-1}$ to all of its neighbours. Since $\widehat{\mathcal{P}}_{v,\ell-1}$ is a minimal $(k-\ell+1)$-representative family for $\mathcal{P}_{v,\ell-1}$, and $\mathcal{P}_{v,\ell-1}$ consists of sets of size ℓ, we have that $|\widehat{\mathcal{P}}_{v,\ell-1}| \leq \binom{k+1}{\ell-1} \leq 2^{k+1}$. In particular, $\widehat{\mathcal{P}}_{v,\ell-1}$ can be encoded using $O\left(k\binom{k+1}{\ell-1} \log n\right)$ bits, and broadcasting it to all neighbours can be done in $O\left(k\binom{k+1}{\ell-1}\right)$ rounds.
3. Each node $v \in V$, after having received $\widehat{\mathcal{P}}_{u,\ell-1}$ for each $u \in N(v)$, locally constructs the set

$$\mathcal{P}'_{v,\ell} = \bigcup_{u \in N(v)} \left\{\{v\} \cup U : U \in \widehat{\mathcal{P}}_{u,\ell-1} \text{ and } v \notin U\right\}.$$

   We observe that $\mathcal{P}'_{v,\ell}$ is $(k-\ell)$-representative for $\mathcal{P}_{v,\ell}$. If $W \subseteq V$ is a set of size $k-\ell$ that does not intersect some $U \in \mathcal{P}_{v,\ell}$, then there is some $u \in N(v)$ and $U' \in \mathcal{P}_{u,\ell-1}$ such that $W \cup \{v\}$ does not intersect U'. Since $\widehat{\mathcal{P}}_{u,\ell-1}$ is $(k-\ell+1)$-representative for $\mathcal{P}_{u,\ell-1}$, there is $R \in \widehat{\mathcal{P}}_{u,\ell-1}$ such that R does not intersect $W \cup \{v\}$, and thus $R \cup \{v\} \in \mathcal{P}'_{v,\ell}$ does not intersect W. Thus, computing a minimal $(k-\ell)$-representative family $\widehat{\mathcal{P}}_{v,\ell}$ for $\mathcal{P}'_{v,\ell}$ gives a $(k-\ell)$-representative family for $\mathcal{P}_{v,\ell}$ by Lemma 3.

Clearly, there is a k-path terminating at v if and only if $\widehat{\mathcal{P}}_{v,k}$ is non-empty. There are k phases corresponding to $\ell = 1, 2, \ldots, k$ in the algorithm, and each of these phases runs in

$O\big(k\binom{k+1}{\ell-1}\big)$ rounds, where the hidden constant does not depend on ℓ. Since it holds that is

$$\sum_{\ell=1}^{k} k\binom{k+1}{\ell-1} = k\sum_{\ell=1}^{k}\binom{k+1}{\ell-1} \leq k2^k\,,$$

the total running time is $O(k2^k)$.

Finally, let us observe that it is easy to modify this algorithm to *find* a k-path, in the sense that each node $v \in V$ that is an endpoint of a k-path has full knowledge of a single k-path: we annotate each set $U \in \widehat{\mathcal{P}}_{v,\ell}$ with a 'witness' ℓ-path with node set U terminating at v. This can be done without increasing the asymptotic communication cost.

▶ **Theorem 4.** *Finding k-paths can be done in $O(k2^k)$ rounds in the broadcast* CONGEST *model.*

Finding cycles. We first note that it is easy to adapt the k-path algorithm to detect k-cycles that contain a fixed node $w \in V$. We modify the definition of $\mathcal{P}_{v,\ell}$ to require that the paths have w as a starting point; otherwise the algorithm proceeds as before. If any neighbour of w detects a $(k-1)$-path starting from w, then there is a k-cycle containing w; likewise, if there is a k-cycle containing w, then some neighbour of w will detect a $(k-1)$-path starting from w. Running this cycle detection algorithm for all choices of the starting node in parallel gives allows us to detect k-cycles in time $O(k2^k n)$:

▶ **Theorem 5.** *In the broadcast* CONGEST *model,*
1. *finding k-cycles containing a fixed node $w \in V$ can be done in $O(k2^k)$ rounds, and*
2. *finding k-cycles can be done in $O(k2^k n)$ rounds.*

Finding trees. We now show how to extend the path detection algorithm to any target graph H that is acyclic. Let H be a tree on k nodes. Fix an arbitrary node of H as a root, and identify the nodes of H with $1, 2, \ldots, k$ so that if i is a descendant of j then $j > i$. In particular, node k is the root. Denote by $H[i]$ the subtree of H rooted at node i, and denote by s_i the number of nodes in $H[i]$.

For $v \in V$ and $i \in \{1, 2, \ldots, k\}$, we define the set family

$$\mathcal{T}_{v,i} = \big\{U \subseteq V\colon \text{there is a copy of } H[i] \text{ with node set } U \text{ and root } v\big\}\,.$$

Again, if we know $\mathcal{T}_{u,j}$ for all $u \in N(v)$ and and $j < i$, we can compute $\mathcal{T}_{v,i}$. Let $c_1, c_2, \ldots, c_\ell$ be the children of i in H. Then $\mathcal{T}_{v,i}$ contains exactly the sets of form

$$\{v\} \cup U_1 \cup U_2 \cup \cdots \cup U_\ell\,,$$

where $U_x \in \mathcal{T}_{u,c_x}$ for some $u \in N(v)$ and $U_x \cap U_y = \emptyset$ for all $x \neq y$.

Similarly to the path algorithm, for $i = 1, 2, \ldots, k$, we iteratively construct $(k - s_i)$-representative family $\widehat{\mathcal{T}}_{v,i}$ for $\mathcal{T}_{v,i}$:
1. If i is a leaf in H, then $\widehat{\mathcal{T}}_{v,i} = \mathcal{T}_{v,i} = \{\{v\}\}$.
2. If i is an internal node in H with children $c_1, c_2, \ldots, c_\ell$, then we construct $\mathcal{T}'_{v,i}$ by taking all sets of form

   $$\{v\} \cup U_1 \cup U_2 \cup \cdots \cup U_\ell\,,$$

   where $U_x \in \widehat{\mathcal{T}}_{u,c_x}$ for $u \in N(v)$ and $U_x \cap U_y = \emptyset$ for all $x \neq y$, and compute a minimal $(k - s_i)$-representative family $\widehat{\mathcal{T}}_{v,i}$ for $\mathcal{T}'_{v,i}$.

We observe that after step i, each node v knows if there is a copy of $H[i]$ that is rooted at v, which implies the correctness of the algorithm. Time complexity follows by the same arguments as in the path detection algorithm. To summarise:

▶ **Theorem 6.** *For any tree H on k nodes, H-subgraph detection can be done in $O(k2^k)$ rounds in the broadcast* CONGEST *model.*

Finding pseudotrees. Recall that a pseudotree is a connected graph that has at most one cycle, that is, a graph that consist of a tree and at most one additional edge. Let H be a pseudotree on k vertices, and let us assume that it has a cycle. Let $e = \{u_1, u_2\}$ be an arbitrary edge on the cycle.

We slightly modify our tree detection algorithm to find a copy of H as follows. Consider the tree H' obtained from H by deleting the edge e. We identify the nodes of H' with $1, 2, \ldots, k$ as before, using u_2 as the root node k; let us assume that u_1 is identified with j.

The basic idea is to modify the tree detection algorithm to keep track which nodes of the input graph can play the role of j in a copy of H'. Specifically, when constructing the sets $\widehat{\mathcal{T}}_{v,i}$, we keep track of which node plays the role of j in the copies of $H'[i]$ we have found so far. This requires the following changes to the tree detection algorithm:

- For i that is not on the unique path from j to k, no changes are required.
- For the step corresponding to j, each node $v \in V$ performs the step as before, but marks itself as the node playing the role of j when broadcasting the constructed representative family $\widehat{\mathcal{T}}_{v,j}$.
- For i that is on the path from j to k, each node $v \in V$ keeps track of $\widehat{\mathcal{T}}_{v,i}$ separately for each possible choice of j it sees.

After the algorithm is finished, all nodes check if they see a copy of H' where the node j is one of their neighbours, which happens if and only if a copy of H exists in the graph. At each step, each node has to broadcast at most n copies of the representative family, giving us the following:

▶ **Theorem 7.** *For any pseudotree H on k nodes, H-subgraph detection can be done in $O(k2^k n)$ rounds in the broadcast* CONGEST *model.*

5 Enumerating cliques and short cycles in degenerate graphs

Acyclic orientations with bounded outdegree. Our algorithms for enumerating cliques and cycles in sparse graphs exploit the fact that degenerate graphs have acyclic orientations with bounded outdegree. Once we have such an orientation, the nodes can co-ordinate how to distribute the information about the edges present in the input graph, which avoids having too much congestion over a single communication link.

For any $\alpha \in \mathbb{N}$, we say that σ is an α-bounded orientation [10] of $G = (V, E)$ if

1. every node $v \in V$ has $\mathrm{outdeg}_\sigma(v) \leq \alpha$, and
2. the orientation σ is acyclic.

For brevity, we call such orientations simply α-orientations. The class of degenerate graphs can be characterised by the existence of such orientations.

▶ **Lemma 8.** *A graph G is d-degenerate if and only if there exists a d-orientation of G.*

Barenboim and Elkin [2] gave efficient distributed algorithms for computing such orientations under the name Nash–Williams forests-decompositions [25]. For the sake of completeness, we formulate more or less the same algorithm here. Barenboim and Elkin [2]

also show how to compute such orientations even without knowing either the degeneracy, or alternatively, the number of nodes in the network in $O(\log n)$ rounds using messages of size $O(\log n)$.

▶ **Lemma 9.** *If G is d-degenerate, then it has at most nd edges.*

Proof. Consider the d-orientation given by Lemma 8. Counting the outgoing edges yields

$$|E| = \sum_{v \in V} \text{outdeg}(v) \leq nd.$$ ◀

Let G be a d-degenerate graph and $C > 2$ be a constant. For all $i \geq 0$, define iteratively the node sets $V_0, V_1, V_2, \ldots$ as follows:

$$V_0 = V\,,$$
$$V_{i+1} = \{v \in V_i : \deg(v) > Cd\}\,.$$

We note that each node can compute $i_v = \max\{i \in \mathbb{N} \colon v \in V_i\}$ in i_v rounds.

▶ **Lemma 10.** *We have that $|V_{i+1}| \leq 2/C \cdot |V_i|$.*

Proof. Suppose the opposite holds, that is, there are $h > 2/C \cdot |V_i|$ nodes $v \in V_i$ with $\deg(v) > Cd$. As $G[V_i] \subseteq G$ is d-degenerate, counting the number of edges incident to these h nodes yields that there are at least

$$\frac{Cdh}{2} > \frac{2 \cdot |V_i| Cd}{2C} = |V_i| \cdot d$$

edges, which contradicts Lemma 9. ◀

By the above lemma, we get that every iteration we lose a constant fraction $1/c$ of nodes, where $c = C/2$. Thus, $|V_i| \leq n/c^i$ and after $r = \log n / \log c + 1 = \log_c n$ iterations we have

$$|V_r| = \frac{n}{c^{\log_c n + 1}} \leq 1.$$

We can now compute an acyclic orientation of G using the above scheme. Each node $v \in V_i \setminus V_{i+1}$ removed in iteration i orients any edge $\{v, u\}$ with $u \in V_{i+1}$ towards u. Clearly there will be at most Cd edges pointing away from v.

▶ **Lemma 11** ([2])**.** *Let G be a d-degenerate graph. For any constant $\varepsilon > 0$, we can compute a $(2d + \varepsilon)$-orientation of G in $O(\log n)$ rounds in the broadcast* CONGEST *model.*

Enumerating cliques. We start with a k-clique enumeration algorithm that works in d-degenerate graphs for any $k \leq d$. Note that the scenario $k > d$ is fatuous, as a d-degenerate graph cannot contain a clique with more than d nodes. Let $G = (V, E)$ be the input graph. The algorithm is as follows:

1. Compute the α-orientation of G for $\alpha \in \Theta(d)$.
2. Each $v \in V$ broadcasts the endpoints $N_{\text{out}}(v)$ of outgoing edges to all its neighbours.
3. Each $v \in V$ locally constructs the induced subgraph $G[F]$, where

$$F = \{\{u, w\} : u \in N^+(v), w \in N_{\text{out}}(u)\}.$$

4. Each $v \in V$ outputs all k-cliques in $G[F]$.

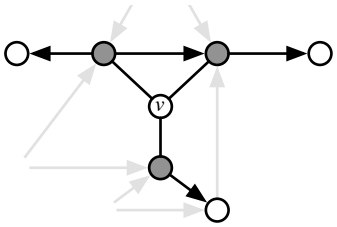


Figure 1 Example of the information gathered by the clique detection algorithm. Once an α-orientation of the graph has been computed, the algorithm ensures that node v obtains all outgoing edges of its neighbours (black edges). Neighbours of v are are dark gray and the edges not observed by v are gray. Here, node v detects the triangle its part of.

Figure 1 illustrates the information gathered by the above algorithm.

In order to analyse the algorithm, first observe that the algorithm clearly only outputs k-cliques that are present in G. We now argue that if G contains a k-clique, then some node will list it in the output. Let $K \subseteq V$ be a k-clique in G. Note that $G[K]$ must contain a sink node $v \in K$ with respect to the α-orientation σ: if such a sink node does not exist, then every node in $G[K]$ would have positive outdegree implying that σ is not acyclic, which is absurd. Hence, the sink node $v \in K$ will receive the set $N_{\text{out}}(u)$ from every $u \in K$. In particular, we have

$$K \subseteq \bigcup_{u \in N(v)} N_{\text{out}}(u).$$

Therefore, the sink node v detects the clique K in the subgraph H it constructs in the third step, and thus, will list it in its output.

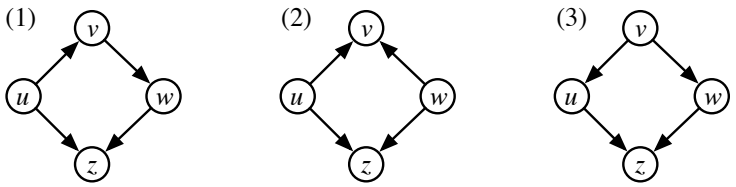
It remains to analyse the time complexity of the algorithm. By Lemma 11, the first step takes $O(\log n)$ rounds. The second step takes $O(\alpha)$ rounds, as broadcasting a single identifier requires $O(\log n)$ bits to be transmitted along each edge. Since there are at most α such identifiers to be broadcast (as there are at most α outgoing edges) and $\alpha \in O(d)$, we get that this step takes $O(d)$ time. The remaining steps require no communication, and hence, all nodes declare their output within $O(\log n + d)$ rounds.

▶ **Theorem 12.** *Let $k \leq d$. In any d-degenerate graph G, enumerating k-cliques takes $O(d + \log n)$ rounds in the broadcast* CONGEST *model.*

As triangles are 3-cliques, we immediately get the following corollary.

▶ **Corollary 13.** *In any d-degenerate graph G, enumerating triangles takes $O(d + \log n)$ rounds in the broadcast* CONGEST *model.*

Enumerating 4-cycles. In the case of 4-cycles, it turns out that we can use the same algorithm as for enumerating cliques, except we check for the existence of a 4-cycle in the locally constructed subgraph $G[F]$ instead of a k-clique. Let $C = \{u, v, w, z\}$ be a 4-cycle in G. As before, we first obtain α-orientation σ of the d-degenerate input graph G using Lemma 11. Since the orientation is acyclic, the cycle graph $G[C]$ must have at least one source and a sink in the orientation. In particular, there are three possible cases (up to isomorphisms):



To show that some node detects the cycle C, we consider each of these three cases. In each case, observe that u sends v the set $N_{\text{out}}(u)$ and w sends the set $N_{\text{out}}(w)$ to v. Thus, node v learns about the edges $\{u, z\}$ and $\{w, z\}$ as $z \in N_{\text{out}}(u) \cap N_{\text{out}}(w)$. Since v trivially knows about the edges $\{u, v\}$ and $\{v, w\}$, we get that in each case node v learns about all edges in C and lists C in its output. Note that if we want that each 4-cycle is listed by exactly one node, the nodes can detect if multiple nodes would be listing the same cycle C and output C only if they are the node with smallest identifier having knowledge about it.

As the algorithm communicates the same information as the clique detection algorithm given before, we get that the time complexity of detecting 4-cycles is the same.

▶ **Theorem 14.** *In any d-degenerate graph G, enumerating 4-cycles takes $O(d+\log n)$ rounds in the broadcast* CONGEST *model.*

Enumerating 5-cycles. In order to enumerate 5-cycles, we use the same underlying idea, but now nodes also communicate directed outgoing paths of length 2 instead of just single outgoing edges. With this modification in mind, the algorithm is as follows:

1. Compute the α-orientation of G for $\alpha \in \Theta(d)$.
2. Each $v \in G$ broadcasts its endpoints $N_{\text{out}}(v)$ of outgoing edges to all its neighbours.
3. Each $v \in G$ broadcasts the set of outgoing length-2 paths

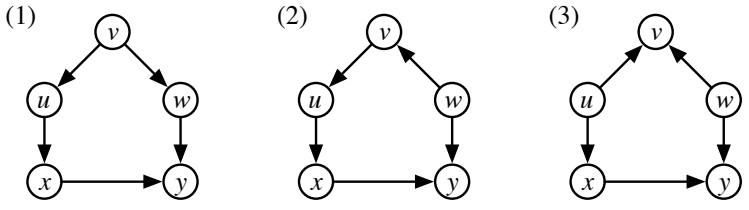
$$L(v) = \{\{u, w\} : u \in N_{\text{out}}(v), w \in N_{\text{out}}(u)\}$$

   to all its neighbours.
4. Each $v \in G$ locally constructs the subgraph $G[F]$, where

$$F = \big\{\{u, w\} : u \in N^+(v), w \in N_{\text{out}}(u)\big\} \cup \bigcup_{u \in N_{\text{out}}(v)} L(u).$$

5. Each $v \in V$ outputs all 5-cycles in $G[F]$.

Let $C = \{u, v, w, x, y\}$ be a 5-cycle in G. To see that the algorithm outputs the cycle C, it suffices to show that some node $v \in C$ learns about all the edges in C. As the α-orientation is acyclic, the cycle C can be oriented in the following three ways (up to isomorphisms):



To see that this is the case, one can readily observe that the length of the longest directed path in an acyclically oriented 5-cycle has to be either 2, 3 or 4, and the length of this path determines the orientation of all other edges.

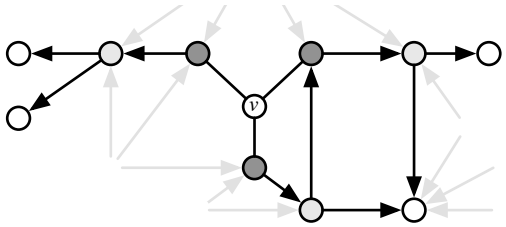


Figure 2 Example of information the nodes gather when executing the 5-cycle detection algorithm. Once an α-orientation of the graph has been computed, the algorithm ensures that node v obtains all outgoing length-1 and length-2 paths (black directed edges) of its neighbours. Neighbours of v are are dark gray, nodes at distance two are light gray and nodes at distance three are white.

We argue that node v detects the cycle C in all cases. Observe that node x broadcasts $N_{\text{out}}(x)$ to u in the second step and node u broadcasts $N_{\text{out}}(u)$ and $L(u)$ to v in the second and third steps, respectively. Since $y \in N_{\text{out}}(x)$, we have $\{x, y\} \in L(u)$ and $x \in N_{\text{out}}(u)$. Therefore, v learns about the path (u, x, y). Since w broadcasts $N_{\text{out}}(w)$ to v, node v also learns about the edge $\{w, y\}$. Since node v can trivially detect the edges $\{u, v\}$ and $\{v, w\}$, it follows that node v detects the cycle C in the final step.

The time complexity of the first step is again $O(\log n)$. The second step consists of broadcasting the set of up to α identifiers, which takes $O(\alpha)$ rounds as before. In the third step, each node $v \in G$ broadcasts $L(v)$ which may contain up to $O(\alpha^2)$ edges. Thus, the third step takes $O(\alpha^2)$ rounds. No communication occurs in the final steps of the algorithm, which yields that the total time complexity of the algorithm is $O(\log n + \alpha^2)$. As we can choose $\alpha \in \Theta(d)$, we obtain the following result.

▶ **Theorem 15.** *In any d-degenerate graph G, enumerating 5-cycles takes $O(d^2 + \log n)$ rounds in the broadcast* CONGEST *model.*

6 Lower bounds

Lower bounds for long cycles. We first prove our lower bounds for the detection of long cycles:

▶ **Theorem 16.** *In the* CONGEST *model,*

1. *finding k-cycles for even k requires $\Omega(n^{1/2}/\log n)$ rounds, and*
2. *there is no algorithm for finding k-cycles for $k \geq 6$ that runs in $O(f(d)n^{\delta})$ rounds on d-degenerate graphs for any function f and $\delta < 1/2$.*

Proof. We use a slight modification of the cycle detection lower bound construction of Drucker et al. [13]; that is, we prove the lower bound by a reduction to known set disjointness lower bounds in two-player communication complexity.

Specifically, given sets A and B over M-element universe, we construct a n-node graph $G_{A,B} = (V_A \cup V_B, E)$ such that (1) $G[V_A]$ depends only on A, (2) $G[V_B]$ depends only on B, (3) there are at most C edges crossing the cut (V_A, V_B), and (4) $G_{A,B}$ contains a cycle of length k if and only if A and B are non-disjoint. This allows us to use any k-cycle detection to solve 2-party set disjointness; Alice simulates all nodes in V_A, Bob simulates nodes in V_B, and messages over edges crossing the cut (V_A, V_B) are sent between players. By known lower bounds, $\Omega(M)$ bits have to be sent over the cut, giving lower bound of $\Omega(M/C \log n)$ rounds in the CONGEST model.

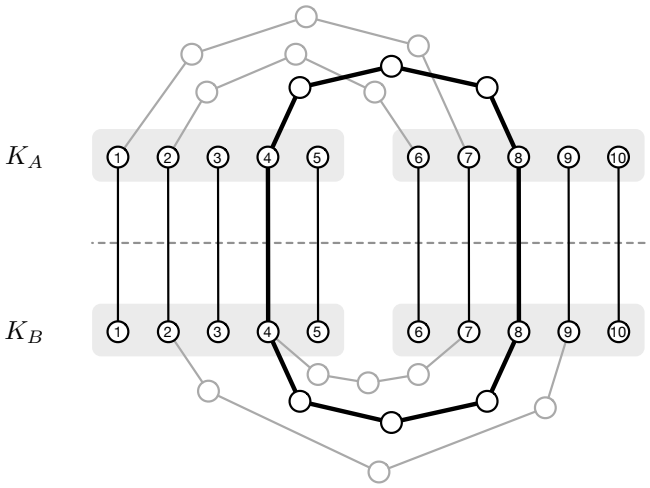


Figure 3 Example of the lower bound construction for $k = 8$, $N = 5$ and $M = 25$. A 8-cycle is highlighted in black; note that a 8-cycle can occur only when the corresponding paths are present both in K_A and K_B.

We show that for any $k \geq 6$ and all $N \in \mathbb{N}$, there is a construction satisfying the above with parameters $M \in \Theta(N^2)$, $n \in \Theta(N^2)$ and $C = 2N$, with the additional property that any graph obtained from this construction is 2-degenerate. This implies both of the claims.

Let $k \geq 6$ and $N \in \mathbb{N}$ be fixed, and let $A, B \subseteq 2^{[N^2]}$ be a set disjointness instance. Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$. We take two disjoint copies K_A and K_B of a complete bipartite graph $K_{N,N}$ on $2N$ nodes; assume that the nodes of the both copies are labelled in a consistent manner with integers $1, 2, \ldots, 2N$, and that the edges are likewise consistently labelled with the elements of $2^{[N^2]}$. We now finish the construction as follows:

1. We connect nodes labelled with the same integer in K_A and K_B with an edge.
2. We remove each edge in K_A if the corresponding element in $2^{[N^2]}$ is not in A, and we remove each edge in K_B if the corresponding element in $2^{[N^2]}$ is not in B.
3. We replace all remaining edges in K_A with $(\ell_1 - 1)$-paths, and all remaining edges in K_B with $(\ell_2 - 1)$-paths.

It is now easy to verify that there are no k-cycles remaining inside either K_A or K_B, and there is a k-cycle if and only if $A \cap B \neq \emptyset$. The final graph has at most $(k-4)N^2 + 4N$ nodes and at most $(k-2)N^2 + 2N$ edges; the cut between K_A and K_B has size $2N$. Finally, it is easy to see that the graph has degeneracy 2. For each path added in step (3), we pick an internal node and orient all edges away from it, and orient the cut edges between K_A and K_B arbitrarily; the resulting orientation has maximum outdegree two. ◀

Lower bounds for short cycles. We observe that the lower bounds given by Drucker et al. [13] imply that finding 4-cycles and 5-cycles in d-degenerate graphs requires $\Omega(d/\log n)$ rounds: a careful examination shows that their lower bound graph for 4-cycles has degeneracy $\Theta(n^{1/2})$ and the lower bound graph for 5-cycles has degeneracy $\Theta(n)$. Thus, we have the following:

▶ **Theorem 17.** *Finding 4-cycles and 5-cycles in d-degenerate graphs requires $\Omega(d/\log n)$ rounds in the* CONGEST *model.*

7 Extensions to the supported CONGEST

The supported CONGEST model. Finally, we discuss how our results can be extended to the supported CONGEST proposed by Schmid and Suomela [29] for studying distributed algorithms in the context of *software defined networking* (SDN).

In the supported CONGEST model, the underlying physical network topology is represented by a *support graph* $G = (V, E)$ and the actual *input graph* H for the distributed algorithm is an arbitrary subgraph of the support. The input graph $H = (U, F)$ inherits the identifiers of the support graph and can be seen as the current logical state of the network. Each node v in the network G is aware of the full topological information about G, but node v has only local information about the current logical state of the network, that is, the actual input graph H. That is, initially node v will know only which of the edges incident to it in G are also present in H. Note that supported CONGEST can be seen as a relaxation of the congested clique model: in the case of the congested clique model, the support graph is a clique.

While many symmetry breaking problems are trivial in the supported CONGEST model, the model remains interesting from the perspective of subgraph detection. Even though all nodes will a priori know that any edge $\{u, v\}$ not present in G will also not be in H, only nodes u and v initially know whether $\{u, v\} \in F$ as well. Thus, the difficulty now becomes verifying which of the possible subgraphs in G remain in H.

Subgraph enumeration in sparse support graphs. We can modify the algorithms given in Section 5 to run faster in the supported CONGEST model, in the case where the support graph has bounded degeneracy. First, all nodes can locally determine the degeneracy of the support graph G and compute (the same) d-orientation σ of G without any communication. The orientation σ restricted to the input graph H will also be an d-orientation of the input H. This saves the additive $O(\log n)$ term in the running time. Moreover, when the nodes communicate their outgoing edges, as every node $v \in V$ knows which outgoing edges of σ *can be* present incident to $u \in V$, it suffices that each node u broadcasts d-length bit string encoding which edges of G are present in the input H. With these modifications in mind, we get the following result.

▶ **Theorem 18.** *In any d-degenerate support graph G in the supported* CONGEST *model,*

1. *enumerating k-cliques can be done in $O(d/\log n)$ rounds for any k,*
2. *enumerating 4-cycles can be done in $O(d/\log n)$ rounds, and*
3. *enumerating 5-cycles can be done in $O(d^2/\log n)$ rounds.*

In particular, in the case that the support graph has degeneracy $O(\log n)$, we can enumerate cliques and 4-cycles in constant number of rounds in the supported CONGEST model.

Cycle detection lower bounds. Finally, we note that all lower bounds from the present work as well as those by Drucker et al. [13] hold in the supported CONGEST. All these lower bounds are obtained by starting from a fixed base graph G, and removing certain edges to obtain a graph G' that encodes a set disjointness instance. Taking the base graph G as the support graph and G' as the input graph yields identical lower bounds for the supported CONGEST.

8 Conclusions

Subgraph detection. We note that there still remain major open questions regarding subgraph detection in the CONGEST model:

- What is the complexity of general subgraph detection? In particular, can the subgraph detection problem solved in linear number of rounds for any constant-size target graph H, or are the target graphs that require a superlinear number of rounds? Note that k-cliques, which appear to be the most difficult case for centralised subgraph detection, can be trivially detected in $O(n)$ rounds for any k.
- What is the precise complexity of detecting even-length cycles? Is there an algorithm matching the $\tilde{\Omega}(n^{1/2})$ lower bound we give, or can this lower bound be improved?

Fixed-parameter techniques. We remark that the many algorithmic techniques from fixed-parameter cycle and path detection algorithms seem to translate very naturally to the distributed limited bandwidth setting, as they effectively either (a) partition the task at hand into multiple independent instances of an easier task (e.g. colour-coding), or (b) compress the intermediate results of the computation (e.g. representative families). As noted before, the seminal colour-coding technique of Alon et al. [1] has been used by various authors for distributed algorithms [9, 16, 17]; indeed, the results in the present work can also be derived via colour-coding, albeit with slightly worse dependence on k, by an easy modification of the congested clique cycle detection algorithm of Censor-Hillel et al. [9]. We also expect that the $O(k2^k)$ round running time for path detection could be improved with randomisation, for example by using algebraic *sieving* techniques [6].

More generally, such fixed-parameter techniques are applicable in the centralised setting beyond subgraph detection, and we expect this to be the case also in the distributed setting. For example, it seems likely that constant-round distributed algorithms for the *graph motif* problem can be obtained either via colour-coding or algebraic sieving [6, 5]. Indeed, this general line of research may even have practical relevance, as evidenced by the efficient parallel implementation of fixed-parameter graph motif algorithms by Björklund et al. [7].

Acknowledgements. We thank Juho Hirvonen, Dennis Olivetti, Rotem Oshman, Chris Purcell, Stefan Schmid and Jukka Suomela for valuable comments and discussions.

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

2 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010. doi:10.1007/s00446-009-0088-2.

3 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

4 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, 2016. doi:10.1145/2903137.

5 Nadja Betzler, Michael R. Fellows, Christian Komusiewicz, and Rolf Niedermeier. Parameterized algorithms and hardness results for some graph motif problems. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008)*, pages 31–43. Springer, 2008.

6 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017. doi:10.1016/j.jcss.2017.03.003.

7 Andreas Björklund, Petteri Kaski, Lukasz Kowalik, and Juho Lauri. Engineering motif search for large graphs. In Ulrik Brandes and David Eppstein, editors, *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 104–118. SIAM, 2015. doi:10.1137/1.9781611973754.10.

8 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast distributed algorithms for testing graph properties. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2016. doi:10.1007/978-3-662-53426-7_4.

9 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2015)*, pages 143–152, 2015.

10 Marek Chrobak and David Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theor. Comput. Sci.*, 86(2):243–266, 1991. doi:10.1016/0304-3975(91)90020-3.

11 Danny Dolev, Christoph Lenzen, and Shir Peled. "tri, tri again": Finding triangles and small subgraphs in a distributed setting - (extended abstract). In Marcos K. Aguilera, editor, *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, volume 7611 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 2012. doi:10.1007/978-3-642-33651-5_14.

12 Rodney G. Downey and Michael R. Fellows. Parameterized computational feasibility. In *Feasible Mathematics II*, pages 219–244, 1994. doi:10.1007/978-1-4612-2566-9_7.

13 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376. ACM, 2014. doi:10.1145/2611462.2611493.

14 P. Erdős, A. Hajnal, and J. W. Moon. A problem in graph theory. *The American Mathematical Monthly*, 71(10):1107–1110, 1964. doi:10.2307/2311408.

15 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Dennis Olivetti Pedro Montealegre, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three notes on distributed property testing. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, 2017.

16 Guy Even, Reut Levi, and Moti Medina. Faster and simpler distributed algorithms for testing and correcting graph properties in the CONGEST-model, 2017. arXiv:1705.04898 [cs.DC].

17 Orr Fischer, Tzlil Gonen, and Rotem Oshman. Distributed property testing for subgraph-freeness revisited, 2017. arXiv:1705.04033 [cs.DS].

18 Fedor V. Fomin, Daniel Lokshtanov, and Saket Saurabh. Efficient computation of representative sets with applications in parameterized and exact algorithms. In *25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, pages 142–151, 2014.

19 Pierre Fraigniaud, Pedro Montealegre, Dennis Olivetti, Ivan Rapaport, and Ioan Todinca. Distributed subgraph detection, 2017. arXiv:1706.03996 [cs.DC].

20 Pierre Fraigniaud, Ivan Rapaport, Ville Salo, and Ioan Todinca. Distributed testing of excluded subgraphs. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceed-*

ings, volume 9888 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2016. doi:10.1007/978-3-662-53426-7_25.

21 Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 381–389. ACM, 2017. doi:10.1145/3087801.3087811.

22 Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST, 2017. arXiv:1705.10195 [cs.DC].

23 Dániel Marx. A parameterized view on matroid optimization problems. *Theoretical Computer Science*, 410(44):4471–4479, 2009.

24 Burkhard Monien. How to find long paths efficiently. *North-Holland Mathematics Studies*, 109:239–254, 1985.

25 Crispin Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, s1-39(1):12–12, 1964. doi:10.1112/jlms/s1-39.1.12.

26 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Tight bounds for distributed graph computations, 2016. arXiv:1602.08481 [cs.DC].

27 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.

28 Oleg Pikhurko. A note on the Turán function of even cycles. *Proceedings of the Americal Mathematical Society*, 140:3687–3692, 2012. doi:10.1090/S0002-9939-2012-11274-2.

29 Stefan Schmid and Jukka Suomela. Exploiting locality in distributed SDN control. In Nate Foster and Rob Sherwood, editors, *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*, pages 121–126. ACM, 2013. doi:10.1145/2491185.2491198.

Distributed Distance-Bounded Network Design Through Distributed Convex Programming*†

Michael Dinitz[1] and Yasamin Nazari[2]

1 **Johns Hopkins University, Baltimore, MD, USA**
`mdinitz@cs.jhu.edu`
2 **Johns Hopkins University, Baltimore, MD, USA**
`ynazari@jhu.edu`

Abstract

Solving linear programs is often a challenging task in distributed settings. While there are good algorithms for solving packing and covering linear programs in a distributed manner (Kuhn et al. 2006), this is essentially the only class of linear programs for which such an algorithm is known. In this work we provide a distributed algorithm for solving a different class of convex programs which we call "distance-bounded network design convex programs". These can be thought of as relaxations of network design problems in which the connectivity requirement includes a distance constraint (most notably, graph spanners). Our algorithm runs in $O((D/\epsilon)\log n)$ rounds in the $\mathcal{LOCAL}$ model and with high probability finds a $(1+\epsilon)$-approximation to the optimal LP solution for any $0 < \epsilon \leq 1$, where D is the largest distance constraint.

While solving linear programs in a distributed setting is interesting in its own right, this class of convex programs is particularly important because solving them is often a crucial step when designing approximation algorithms. Hence we almost immediately obtain new and improved distributed approximation algorithms for a variety of network design problems, including Basic 3- and 4-Spanner, Directed k-Spanner, Lowest Degree k-Spanner, and Shallow-Light Steiner Network Design with a spanning demand graph. Our algorithms do not require any "heavy" computation and essentially match the best-known centralized approximation algorithms, while previous approaches which do not use heavy computation give approximations which are worse than the best-known centralized bounds.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases distributed algorithms, approximation algorithms, convex programming

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.5

1 Introduction

Distributed network design is a classical type of distributed algorithmic problem, going back at least to the seminal work on distributed MST by Gallager, Humblet, and Spira [16]. By "network design", we mean the class of problems which can be phrased as "given input graph G, find a subgraph H which has some property P, and minimize the cost of H". Clearly different properties P, and different notions of cost, lead to very different problems. One important class of problems are *distance-bounded* network design problems, where the property P is that certain pairs of vertices are within some distance of each other in H (where distance refers to the shortest-path distance). The most well-known type of distance-bounded

* Supported in part by NSF awards 1464239 and 1535887.
† A full version of this paper is available at `https://arxiv.org/abs/1703.07417`.

network design problems are problems involving *graph spanners*, in which the distance requirement is that the distance in H for all (or certain) pairs is within a certain factor (known as the stretch) of their original distance in H. But there are many other important versions of distance-bounded network design, such as the bounded diameter problem [12] and the shallow-light Steiner tree/network problems [19].

Many of these problems are NP-hard, so they cannot be solved optimally in polynomial time even in the centralized setting unless P=NP. Thus they have been studied extensively from an approximation algorithms point of view, where we design algorithms which approximate the optimal solution but which run in polynomial time. For many of these problems, a key step in the best-known centralized approximation algorithm is solving a linear programming relaxation of the problem, and then rounding the optimal fractional solution into a feasible integral solution. Interestingly, it is relatively common for the rounding to be "local": if we are in a distributed setting and happen to know the optimal fractional LP solution, then the algorithm used to round this to an integral solution can be accomplished with a tiny amount of extra time (either 0 or a small constant number of rounds). So the bottleneck when trying to make these algorithms distributed is solving the LP, not rounding it.

Solving LPs in distributed settings has received only a small amount of attention, since it unfortunately turns out to be extremely challenging in general. Most notably, Kuhn, Moscibroda, and Wattenhofer [21] gave an efficient distributed algorithm (in the $\mathcal{LOCAL}$ model of distributed computation) for packing/covering LPs. Unfortunately, the LPs used for distance-bounded network design are not packing/covering LPs[1], and hence we are not able to use their techniques. Floréen et al. [15] also studied a special class of linear programs, namely min-max LPs, in distributed settings, which also cannot be used for our problems of interest. In this paper we show how to solve these LPs (and convex generalizations of them) in the $\mathcal{LOCAL}$ model of distributed computation, which almost immediately gives the best-known results for a variety of distance-bounded network design problems.

In particular, for many network design problems (Directed k-Spanner, Basic 3-Spanner, Basic 4-Spanner, Lowest-Degree k-Spanner, Directed Steiner Network with Distance Constraints with spanning demands, and Shallow-Light Steiner Network with spanning demands) we give approximation algorithms which run in $O(D \log n)$ rounds (where D is the maximum distance bound) and have the same approximation ratios as in the centralized setting. Previous distributed algorithms for these problems with similar round complexity have either used "heavy" computations (non-polynomial time algorithms) at the nodes (in which case they can often do *better* than the best computationally-bounded centralized algorithm), or give approximation bounds which are asymptotically worse than the best centralized bounds. See Section 1.2 for more discussion of previous work.

1.1 Our Results

We give two main types of results. First, we give a distributed algorithm that (approximately) solves distance-bounded network design convex programs with small round complexity. We then use this result to (almost immediately) get improved distributed approximation algorithms for a variety of network design problems.

[1] They can be turned into packing/covering LPs through a projection operation, but unfortunately this technique results in an exponential number of constraints, making [21] inapplicable. However, this technique has been used in the centralized setting for the fault-tolerant directed k-spanner problem [9].

1.1.1 Solving convex programs

Stating our main technical result (distributed approximations of distance-bounded network design convex programs) in full generality requires significant technical setup, so we provide an informal description here. See Section 4 for the full definitions and theorem statements (Theorem 13 in particular). But informally, a distance-bounded network design convex program is the following. We are given a graph $G = (V, E)$, a set $\mathcal{S} \subseteq V \times V$, and for each $(u, v) \in \mathcal{S}$ there is a set of "allowed" $u - v$ paths $\mathcal{P}_{u,v}$. Roughly speaking, we typically assume that the allowed paths are short, and define D to be maximum length of such paths. Informally, the integral problem is to find a subgraph H of G so that every $(u, v) \in \mathcal{S}$ is connected by at least one path from $\mathcal{P}_{u,v}$ in H, and the goal is to minimize some notion of "cost". If our notion of "cost" is captured by an objective function $g : \mathbb{R}_{\geq 0}^{|E|} \to \mathbb{R}$ (which is typically linear, but which can be more general convex functions as long as they satisfy a "partitionability" constraint – see Section 4 for the details), then the natural relaxation of this problem is the following convex program, which has a variable x_e for every edge and a variable f_P for every allowed path.

$$
\begin{aligned}
\min \quad & g(x) \\
\text{s.t.} \quad & \sum_{P \in \mathcal{P}_{u,v} : e \in P} f_P \leq x_e && \forall (u, v) \in \mathcal{S}, \forall e \in E \\
& \sum_{P \in \mathcal{P}_{u,v}} f_P \geq 1 && \forall (u, v) \in \mathcal{S} \\
& x_e \geq 0 && \forall e \in E \\
& f_P \geq 0 && \forall (u, v) \in \mathcal{S}, \forall P \in \mathcal{P}_{u,v}
\end{aligned}
$$

Informally, the first type of constraint says that an allowed path is included only if all edges in it are included, and the second type of constraint required us to include at least one allowed path for each $(u, v) \in \mathcal{S}$. We call this type of convex program a *distance-bounded network design convex program*. It is clearly not a packing/covering LP due to the first type of constraint, and hence there is no known distributed algorithm to solve this kind of program. However, note that if the maximum length of any allowed path is constant, then there are only a polynomial number of such paths, and hence the size of the convex program is polynomial and so it can be solved in polynomial time in the centralized setting under reasonable assumptions on g (see [17] for details on solving convex programs in polynomial time).

Our main technical result is that we can approximately solve these optimization problems even in a distributed setting. For any path P let $\ell(P)$ denote the length of the path (the number of edges in it).

▶ **Theorem 1.** *For any constant $\epsilon > 0$, any distance-bounded network design convex program can be solved up to a $(1 + \epsilon)$-approximation in $O(D \log n)$ rounds in the $\mathcal{LOCAL}$ model, where $D = \max_{(u,v) \in \mathcal{S}} \max_{P \in P_{u,v}} \ell(P)$. Moreover, if the convex program can be solved in polynomial time in the centralized sequential setting, then the distributed algorithm uses only polynomial-time computations at every node*

The dependence on ϵ in the above theorem is hidden in the $O(\cdot)$ notation – see Theorem 13 for the full statement.

Our main technique is to use a distributed construction of *padded decompositions*, a specific type of network decomposition which we explain in detail in Section 3. Padded decompositions have been very useful for metric embeddings and approximation algorithms

(e.g., [18, 20]), but to the best of our knowledge have not been used before in distributed algorithms (with the exception of [10], which used a special case of them to give a distributed algorithm for the fault-tolerant 2-spanner problem). Very similar decompositions, such as the famous Linial-Saks decomposition [22], have been used extensively in distributed settings, but the guarantees for padded decompositions are somewhat different (and we believe that these decompositions may prove useful in the future when designing distributed approximation algorithms). In Section 3 we give a distributed algorithm in the $\mathcal{LOCAL}$ model to construct padded decompositions. These padded decompositions allow us to solve a collection of "local" convex programs with the guarantees that a) most of the demands in $\mathcal{S}$ are satisfied in one of the local programs, and b) the solutions of the local convex programs combine into a (possibly infeasible) global solution with cost at most the cost of the global optimum. Then by averaging over $O(\log n)$ of these decompositions we get a feasible global solution which is almost optimal.

1.1.2 Distributed approximation algorithms for network design

Solving convex programming problems in distributed environments is interesting in its own right, and Theorem 1 is our main technical contribution, but the particular class of convex programs that we can solve are mostly interesting as convex relaxations of interesting combinatorial optimization problems. Many of the problems are NP-hard, but there has been significant work (some quite recent) on designing approximation algorithms for them (see, e.g., [9, 6, 11, 5]). Almost all of these approximations depend on convex relaxations which fall into our class of "distance-bounded network design convex programs". This means that as long as the rounding scheme can be computed locally, we can design distributed versions of these approximation algorithms by using Theorem 1 to solve the appropriate convex relaxation and then using the local rounding scheme.

We are able to use this framework to give distributed approximation algorithms for several problems. Most of them are variations of *graph spanners*, which were introduced by Peleg and Ullman [26] and Peleg and Schäffer [25], and are defined as follows.

▶ **Definition 2.** Let $G = (V, E)$ be a graph (possibly directed), and let $k \in \mathbb{N}$. A subgraph H of G is a *k-spanner* of G if $d_H(u, v) \leq k \cdot d_G(u, v)$ for all $u, v \in V$. The value k is called the *stretch* of the spanner.

Before stating our results, we first define the problems. In the Basic k-Spanner problem we are given an undirected graph G and a value $k \in \mathbb{N}$. A subgraph H of G is a feasible solution if it is a k-spanner of G, and the objective is to minimize the number of edges in H. For $k = 3, 4$, the best-known approximation algorithm for this problem is $\tilde{O}(n^{1/3})$ [5, 11]. If the input graph G (and the solution H) are directed, then this is the Directed k-Spanner problem, for which the best-known approximation is $\tilde{O}(\sqrt{n})$ [5]. If the objective is instead to minimize the maximum degree in H then this is the Lowest-Degree k-Spanner problem, for which the best-known approximation is $\tilde{O}(n^{(1-1/k)^2})$ [6].

The following theorem contains our results on distributed approximations of graph spanners. Informally, it states that we can give the same approximations in the $\mathcal{LOCAL}$ model as are possible in the centralized model.

▶ **Theorem 3.** *There are algorithms in the $\mathcal{LOCAL}$ model that w.h.p.[2] provide the following guarantees. For* Directed k-Spanner*, the algorithm runs in $O(k \log n)$ rounds and gives*

[2] By "with high probability" (or w.h.p.), we mean with probability at least $1 - 1/n^c$ for some $c \geq 1$.

an $\tilde{O}(\sqrt{n})$-approximation. For Basic 3-Spanner *and* Basic 4-Spanner*, the algorithms run in $O(\log n)$ rounds and gives an $\tilde{O}(n^{1/3})$-approximation. For* Lowest-Degree k-Spanner*, the algorithm runs in $O(k \log n)$ rounds and gives an $\tilde{O}(n^{(1-1/k)^2})$-approximation. All of these algorithms use only polynomial-time computations at each node.*

We emphasize that our algorithms for these spanner problems both match the best-known centralized approximations and only use polynomial-time computations at each node. There is significant previous work (see Section 1.2) on designing distributed approximation algorithms for these and related problems that has only one of these two properties, but all previous approaches which use only polynomial-time computations necessarily do worse than the best centralized bound (or have much worse round complexity). At a high level, this is because previous approaches (most notably [2]) do not actually use the structure of the centralized algorithm: they only use the efficient centralized approximation as a black box. By going inside the black box and noticing that they all use a similar type of convex relaxation, we can simultaneously get low round complexity, best-known approximation ratios, and efficient local computation.

It turns out that we can use our techniques for an even broader question: Directed Steiner Network with Distance Constraints with a spanning demand graph. In this problem there is a set $\mathcal{S} \subseteq V \times V$ of demands, and for every demand $(u, v) \in \mathcal{S}$ there is a length bound $L(u, v)$. The goal is to find a subgraph H so that $d_H(u, v) \leq L(u, v)$ for all $(u, v) \in \mathcal{S}$, and the objective is to minimize the number of edges in H. The state of the art centralized bound for this problem is a $O(n^{3/5+\epsilon})$-approximation [7], but if we further assume that every vertex $u \in V$ is the endpoint of at least one demand in $\mathcal{S}$ (which we will refer to as a *spanning demand graph*) then it is straightforward to see that the centralized algorithm of [5] for Directed k-Spanner can be generalized to give a $\tilde{O}(\sqrt{n})$-approximation. Our distributed version of this algorithm also generalizes, w.h.p. giving the following result.

▶ **Theorem 4.** *There is an approximation algorithm in the $\mathcal{LOCAL}$ model for* Directed Steiner Network with Distance Constraints *with a spanning demand graph with approximation ratio $\tilde{O}(\sqrt{n})$ which runs in $O((\max_{(u,v)\in\mathcal{S}} L(u, v)) \log n)$ rounds and uses only polynomial-time computations.*

Note that Directed k-Spanner and Basic k-Spanner are special cases of this problem, where there is a demand for every edge and the length bound is just k times the original distance. Interestingly, other network design problems which have proved important for distributed systems are also special cases, including the Distance Preserver problem (when $L(u, v) = d_G(u, v)$ for all $(u, v) \in \mathcal{S}$), the Pairwise k-Spanner problem (where $L(u, v) = k \cdot d_G(u, v)$ for all $(u, v) \in \mathcal{S}$), and the Shallow-Light Steiner Network problem (where $L(u, v) = D$ for all $(u, v) \in \mathcal{S}$, for some global parameter D). Shallow-Light Steiner Network in particular is a key component in state of the art systems for reliable Internet transport [1], although in that particular application the demand graph is not spanning. Extending our techniques to handle totally general demands by giving a distributed version of [7] is an extremely interesting open question.

1.2 Related Work

While distributed solving of convex programs is a natural question, there is little previous work in the $\mathcal{LOCAL}$ model. Possibly most related to our results is a line of work on solving *positive* linear programs (packing and covering LPs). This was introduced by [23], improved by [4], and then essentially optimal upper and lower bounds were given by [21].

Unfortunately, the convex programs we consider are not positive linear programs due to the "capacity" constraints in which some variables appear with positive coefficients while others have negative coefficients.

A special case of our result was proved earlier in [10], who showed how to solve the LP relaxation of Basic 2-Spanner in the $\mathcal{LOCAL}$ model in $O(\log^2 n)$ rounds (they actually show more than this, by giving a distributed algorithm for the *fault-tolerant* version of Basic 2-Spanner, but that is not germane to our results). Our techniques are heavily based on [10], which is itself based on the ideas from [21]. In particular, [21] uses a Linial-Saks decomposition [22] to solve "local" versions of the linear program in different parts of the graph, and then combines these appropriately. To make this work for the Basic 2-Spanner LP relaxation, [10] had to use *padded decompositions*, which can be thought of as a variant of Linial-Saks with slightly different guarantees which, for technical reasons, are more useful for network design LPs. In this paper we extend these techniques further by giving a more general definition of padded decomposition which works for larger distance requirements, showing how to construct them in the $\mathcal{LOCAL}$ model, and then showing that the basic "combining" idea from [10] can be extended to handle these more general decompositions and far more general constraints and objective functions.

The major type of combinatorial optimization problem which our techniques allow us to approximate are various versions of graph spanners. There are an enormous number of papers on spanners in both centralized and distributed models, but fewer papers which attempt to find the "best" spanner for the particular given input graphs (most papers on spanners give existential results and algorithms to achieve them, rather than optimization results). These optimization questions (e.g., Basic k-Spanner, Directed k-Spanner, and Lowest-Degree k-Spanner) have been considered quite a bit in the context of centralized approximation algorithms and hardness of approximation [9, 5, 11, 6, 8], but almost all of the known centralized results use linear programming relaxations, making them difficult to adapt to distributed settings. Hence there have been only two results on optimization bounds in distributed models: [10] and [2].

Barenboim et al. [2] provided a distributed algorithm using Linial-Saks decompositions that for any integer parameters k, α, gives an $O(n^{1/\alpha})$-approximation for Directed k-Spanner in $\exp(O(\alpha)) + O(k)$ time. This is an extremely strong approximation bound, and in fact is better than even the best centralized bound. This is possible due to their use of very heavy (exponential time) local computation. Our algorithms, on the other hand, take polynomial time for local computations. Barenboim et al. [2] show that heavy local computations can be removed from their algorithm by using a centralized approximation algorithm for a variant of spanners known as *client-server k-spanners*, and in particular that an $f(k)$-approximation for client-server k-spanner can be turned into an $O(n^{1/\alpha} f(k))$-approximation algorithm running in $\exp(O(\alpha)) + O(k)$ rounds in the $\mathcal{LOCAL}$ model for minimum k-spanner with only polynomial local computation. So in order to achieve the same asymptotic approximation ratio as the best-known centralized algorithm, the parameter α must be $\Omega(\log n)$ and hence the running time is polynomial in n, even though k might be a constant. It is essentially known (though not written anywhere) that a variety of other results with slightly different tradeoffs can be achieved through similar uses of Linial-Saks [13] or refinements of Linial-Saks such as [14]. However, since all of these approaches treat the centralized approximation algorithm as a black box, none of them can achieve the same approximation ratio as the centralized algorithm without suffering a much worse (usually polynomial) round complexity that the $O(k \log n)$ that we achieve.

2 Preliminaries and Notation

The distributed setting we will be considering is the $\mathcal{LOCAL}$ model [24], in which time passes in synchronous rounds and in each round every node can send an arbitrary message of unbounded size to each of its neighbors in the underlying graph $G = (V, E)$ (as always, we will let $n = |V|$ and $m = |E|$). This is in contrast to the $\mathcal{CONGEST}$ model, where nodes can only send a message of size $O(\log n)$ to each of their neighbors in each round. We are not focusing on the $\mathcal{CONGEST}$ model in this paper. We will assume that all nodes know n (or at least know a constant approximation of n). Usually in this model the communication graph is the same as the graph of computational interest; e.g., we will be trying to compute a spanner of the communication graph itself. But for some applications we will want the graph to be directed, in which case we make the standard assumption that communication is bidirectional: the graph for which we are trying to compute a convex relaxation / network design problem is directed, but messages can be sent in both directions across a link. In other words, the communication graph is just the undirected version of the given directed graph.

For any pair of nodes $u, v \in V$ we define $d(u, v)$ to be the distance between u and v in the communication graph (i.e. the length of a shortest path between u and v regardless of edge directions). We define $B(u, k)$ to be an undirected ball of radius k from u in the communication graph. More precisely, $B(u, k) = \{w \in V \mid d(u, w) \leq k\}$. If x is a vector then we use x_i to denote the i'th component of x. Most of the time our vectors will be indexed by edges in a graph, in which case we will also use the notation $(x_e)_{e \in E}$.

Given a partition of the vertices V of a graph, we will refer to each part of the partition as a "cluster". For any graph $G = (V, E)$ and set $S \subseteq V$, we let $E(S)$ denote the set of edges in the subgraph induced by S, i.e., $E(S) = \{(u, v) \in E \mid u, v \in S\}$. We will frequently need "restrictions" of vectors to induced subgraphs, so for any vector $x \in \mathbb{R}^m$, we define $x^S = (x^S_e)_{e \in E}$ to be the vector in $\mathbb{R}^m$ where $x^S_e = 0$ if $e \notin E(S)$ and $x^S_e = x_e$ if $e \in E(S)$.

3 Padded decompositions

We will now define and give an algorithm to construct *padded decompositions*, which are one of the key technical tools that we will use when designing algorithm to solve distance-bounded network design convex programs. In this section all graphs are undirected and all distances are with respect to this undirected graphs (in fact, the definition and our algorithm work more generally for any metric space). Recall that $B(u, k)$ denotes the undirected ball of radius k from node u (in the communication graph), and $diam(C) = \max_{u,v \in C} d_G(u, v)$, which is often called the weak diameter. Intuitively, a (k, ϵ)-padded decomposition partitions a graph into clusters, where nodes in each cluster are not too far in the original graph, and balls of a radius k are preserved with probability at least $1 - \epsilon$.

▶ **Definition 5.** Given an *undirected* graph G, a (k, ϵ)-padded decomposition, where $0 < \epsilon \leq 1$, is a probability measure μ over the set of graph partitions (clusterings) that has the following properties:

1) For every $P \in \text{supp}(\mu)$,[3] and every cluster $C \in P$, we have: $diam(C) \leq O((k/\epsilon) \log n)$.

2) For every $u \in V$, it holds that $\Pr(\exists C \in P \mid B(u, k) \subseteq C) \geq 1 - \epsilon$. That is to say, the probability that all nodes in $B(u, k)$ are in the same cluster is at least $1 - \epsilon$.

[3] By $\text{supp}(\mu)$ we mean the set of partitions that have non-zero probability.

Algorithm 1: Sampling from a (k, ϵ)-padded decomposition of $G = (V, E)$.

Let $\pi : V \to [n]$ be an arbitrary bijection from V to $[n]$, and let $r = (\frac{2}{\epsilon})k$.
for $v \in V$ **do**
  Sample z_v independently from a distribution with probability density function $p(z_v) = \left(\frac{n}{n-1}\right)\frac{e^{-z_v/r}}{r}$.
  Set the radius $r_v = \min(z_v, r \ln n + k)$.
for $u \in V$ **do**
  Node u joins cluster $C(v)$, such that $d(v,u) \leq r_v \wedge (\pi(v) < \pi(w)\ \forall w \neq v$ s.t. $d(w,u) \leq r_w)$. `// Node` u `joins the cluster` $C(v)$`, with cluster center` v`, which is the first node in the permutation where` $d(v,u) \leq r_v$`.`

This notion of padded decompositions is standard in metric embeddings and approximation algorithms [18, 20], but to the best of our knowledge has not yet been used in distributed algorithms. We first use a centralized algorithm (Algorithm 1) to sample from a (k, ϵ)-padded decomposition, and then describe how it can be implemented in the $\mathcal{LOCAL}$ model. Algorithm 1 and its analysis are similar to a partitioning algorithm proposed in [3], which was shown to have a low probability of separating nodes in a close neighborhood. Due to space constraints, proofs can be found in Appendix A.

For any partition P constructed by Algorithm 1, each cluster is clearly $C(v)$ for some $v \in V$. We call this special node v the *center* of cluster $C(v)$. Later, we will use the center of each cluster for solving locally defined convex programs.

▶ **Lemma 6.** *Algorithm 1 partitions a given undirected graph $G = (V, E)$ into a partition P such that P is sampled from a (k, ϵ)-padded decomposition.*

We will now use an idea similar to the one used in [10] to make Algorithm 1 distributed. In [10] they only considered the special case of $k = 1$ and $\epsilon = 1/2$, which is why we cannot simply use their result as a black box.

▶ **Lemma 7.** *There is an algorithm in the $\mathcal{LOCAL}$ model that runs in $O(\frac{k}{\epsilon}\ln n)$ rounds and samples from a (k, ϵ)-padded decomposition (so every node knows the cluster that it is in).*

Proof. Without loss of generality, we assume that all nodes have unique IDs[4]. The sequence of IDs in ascending order will determine the permutation π used in Algorithm 1, i.e. if $\mathrm{ID}_u < \mathrm{ID}_v$ then $\pi(u) < \pi(v)$. The algorithm proceeds as follows until all nodes have been assigned to a cluster: each node $u \in V$ chooses a radius r_u based on the distribution defined in Algorithm 1. Then every $u \in V$ simultaneously sends a message containing ID_u to all nodes in $B(u, r_u)$. After receiving all the messages, each node chooses the node with the smallest ID as the cluster center. Then Lemma 6 implies that the clusters satisfy the properties of a (k, ϵ)-padded decomposition. Since the radius that each node chooses is $O((k/\epsilon)\log n)$, and each node only communicates with nodes within its radius, the running time in the $\mathcal{LOCAL}$ mode is $O((k/\epsilon)\log n)$. ◀

[4] We assume this since nodes can each draw an ID from a suitably large space, so the probability of a collision is small enough that it does not affect the guarantees required by a padded decomposition.

4 Distributed distance bounded network design convex programming

In this section we prove Theorem 1, giving an algorithm similar to [10] which can almost optimally solve distance-bounded network design convex programs. We first make all definitions formal in Section 4.1, and in particular define formally the class of objective functions where our results hold. Then in Section 4.2 we give a distributed algorithm which solves these programs up to arbitrarily small error. All missing proofs can be found in Appendix B.

4.1 Distance bounded network design convex programs

We will first describe a general class of objective functions that our algorithm applies to. For a graph $G = (V, E)$ and a set $S \subseteq V$, we let $E(S)$ denote the set of edges in the subgraph of G induced by S. Recall that for a vector $x \in \mathbb{R}^m$ (where $m = |E|$), we define $x^S = (x^S_e)_{e \in E} \in \mathbb{R}^m$ to be the vector where $x^S_e = 0$ if $e \notin E(S)$ and $x^S_e = x_e$ if $e \in E(S)$.

▶ **Definition 8.** Given a graph $G = (V, E)$, a function $g : \mathbb{R}^m \mapsto \mathbb{R}$ is *convex partitionable* with respect to G if g is a non-decreasing[5] and convex function with the following property: for all partitions $\sigma = \{\sigma_1, ..., \sigma_\ell\}$ of nodes in V, there exists a non-decreasing function $h_\sigma : \mathbb{R}^\ell \mapsto \mathbb{R}$, s.t. $g(x) = h_\sigma(g(x^{\sigma_1}), g(x^{\sigma_2}), ..., g(x^{\sigma_\ell}))$ for all $x = (x_e)_{e \in E}$ where $x_e = 0$ for any edge e with endpoints in different clusters of σ (equality does not need to hold for vectors x with nonzero values on edges between clusters).

Convex partitionable functions for graphs are a natural class of functions for distributed computing purposes. Moreover, this class includes many types of objective functions that are of interest in network design problems, including p-norms and linear functions. For example, if the function g is the p-norm with $p \in \mathbb{Z}_{\geq 0}$, then it is easy to verify that by setting the function h_σ to also be the p-norm for any partition σ of V, the conditions of Definition 8 are satisfied. Note, since we consider non-negative values, an unweighted sum is just the 1-norm, and the max function is the infinity norm, and hence they will also satisfy the conditions of Definition 8. Similarly, in case of linear functions, it is easy to see that conditions of Definition 8 are satisfied by setting h_σ to be the *unweighted* sum.

There are also other, less trivial examples. For example, it is not hard to show the p-norm of the *degree vector* (rather than just the edge vector) is also convex partitionable with respect to G. An important special case of this is the ∞-norm of the degree vector, i.e., the maximum degree. Since we use this objective in some of our applications (i.e., for the Lowest-Degree k-Spanner problem), we give a short proof of this case in Appendix B. For an integral vector $x \in \mathbb{R}^m$ we can write $g(x) = \max_{v \in V} \deg(v)$. By generalizing this notation to all $x \in \mathbb{R}^m$, we can define fractional node degrees as $\deg(v) = \sum_{u:(v,u) \in E} x_{(v,u)}$[6].

▶ **Lemma 9.** *Given a graph $G = (V, E)$, the function $g(x) = \max_{v \in V}(\sum_{u:(v,u) \in E} x_{(v,u)})$ is convex partitionable w.r.t. G.*

Now that this class of functions has been defined, we can formally define the class of distance-bounded network design convex programs.

[5] Let $f(x_1, ..., x_k)$ be a multivariate function. We will say f is nondecreasing if the following holds: if $x_i \leq x'_i$ for all $1 \leq i \leq k$, then $f(x_1, ..., x_k) \leq f(x'_1, ..., x'_k)$.

[6] Here we are considering out-degree of nodes in a directed graph. It is easy to see that Lemma 9 also holds in cases of in-degree only or sum of out-degree and in-degree. The latter is what are interested in for Section 5.

▶ **Definition 10.** Let $\mathcal{S} \subseteq V \times V$ be a set of pairs in the graph $G = (V, E)$, and for any pair $(u, v) \in \mathcal{S}$ let $\mathcal{P}_{u,v}$ be a set of paths from u to v, which we sometimes call the set of "allowed" paths. Let g be a non-decreasing convex-partitionable function of $x = (x_e)_{e \in E}$ with $g(\vec{0}) = 0$. Then we call a convex program of the following form a *distance bounded network design CP*:

$$\begin{aligned}
\min \quad & g(x) \\
s.t \quad & \sum_{P \in \mathcal{P}_{u,v} : e \in P} f_P \leq x_e && \forall (u,v) \in \mathcal{S}, \forall e \in E \\
& \sum_{P \in \mathcal{P}_{u,v}} f_P \geq 1 && \forall (u,v) \in \mathcal{S} \\
& x_e \geq 0 && \forall e \in E \\
& f_P \geq 0 && \forall (u,v) \in \mathcal{S}, \forall P \in \mathcal{P}_{u,v}
\end{aligned}$$

As we will see in Section 5, many network design problems use linear (or convex) programming relaxations that satisfy the conditions of Definition 10. A key parameter of such a program is the length of the longest allowed path $D = \max_{(u,v) \in \mathcal{S}} \max_{p \in \mathcal{P}_{u,v}} \ell(p)$ (where $\ell(p)$ is the length of path p).

4.2 Distributed Algorithm

In order to solve these convex programs in a distributed manner, we will first use padded decompositions to form a local problem using a simple distributed algorithm. Let P be a partition sampled from a (k, ϵ)-padded decomposition (in particular, obtained by Lemma 7), where $0 < \epsilon \leq 1$. Recall that for each cluster $C \in P$, $E(C) = \{(u, v) \in E \mid u, v \in C\}$. We define $G(C)$ to be the subgraph induced by C. We prove the following lemma in Appendix B.

▶ **Lemma 11.** *For each cluster C sampled from a (k, ϵ)-padded decomposition, there is a distributed algorithm running in $O(\frac{k}{\epsilon} \log n)$ rounds so that every cluster center knows $G(C)$.*

Let $\mathrm{CP}(G)$ be a distance bounded network design CP defined on graph $G = (V, E)$. We will define local convex programs based on a partition P of G that is sampled from a (D, λ)-padded decomposition. The value of $0 < \lambda \leq 1$ will be set later based on the parameters of our distributed algorithm. For each $C \in P$, let $\mathrm{CP}(C)$ be $\mathrm{CP}(G)$ defined on $G(C)$, but where only demands corresponding to any pair $(u, v) \in \mathcal{S}$ in which $B(u, D)$ is fully contained in C are included. We denote the set of these demands by $N(C)$, more precisely, $N(C) = \{(u, v) \in \mathcal{S} \mid B(u, D) \subseteq C\}$. The objective will then be to minimize $g(x) = g\left((x_e)_{e \in E(C)}\right)$. In other words $CP(C)$ is defined as follows:

$$\begin{aligned}
\min \quad & g(x) \\
s.t \quad & \sum_{P \in \mathcal{P}_{u,v} : e \in P} f_P \leq x_e && \forall (u,v) \in N(C), \forall e \in E(C) \\
& \sum_{P \in \mathcal{P}_{u,v}} f_P \geq 1 && \forall (u,v) \in N(C) \\
& x_e \geq 0 && \forall e \in E(C) \\
& f_P \geq 0 && \forall (u,v) \in N(C), \forall P \in \mathcal{P}_{u,v}
\end{aligned}$$

There is a technical subtlety about computing the function g on each cluster, which is the fact that a solution $\langle x^C, f^C \rangle$ of $\mathrm{CP}(C)$ is only defined on $G(C)$. While in practice x^C is a vector defined only on edges in $E(C)$, in our analysis, we will assume that x^C is a vector

Algorithm 2: Distributed algorithm for approximating distance bounded network design CPs.

1 Set $\lambda = \frac{\epsilon(1-\epsilon)}{(2-\epsilon)(1+\epsilon)}$ and $t = \left\lceil \frac{16(1-\frac{\epsilon}{2})(1+\epsilon)\ln n}{\epsilon^2} \right\rceil$.
2 Sample from (D, λ)-padded decompositions t times by Lemma 7, and let P_i be the partition obtained in the i'th run.
3 For each cluster $C \in P_i$, the center of cluster C computes $G(C)$ (see Lemma 11).
4 The center of each cluster $C \in P_i$ solves $CP(C)$ and sends the solution $\langle x^{C,i}, f^{C,i} \rangle$ to all nodes $u \in C$.
5 **for** $e = (u, v) \in E$ **do**
6 Let $I_{u,v} = \{i \mid \exists C \in P_i : u, v \in C\}$.
`// these are the iterations in which both endpoints are in same cluster`
7 $\tilde{x}_e \leftarrow \min(1, \frac{1+\epsilon}{t} \sum_{i \in I_{u,v}} x_e^{C_{u,i},i})$.

in $\mathbb{R}^{|E|}$ and $x_e^C = 0$ for all $e \notin E(C)$. This assumption does not impact the correctness of algorithm. The following lemma is similar to Lemma 3.8 in [10], and we show that it holds for our modified definition of local convex programs and for generalized objective functions that satisfy Definition 8.

▶ **Lemma 12.** *Let $\langle x^*, f^* \rangle$ be an optimal solution of $CP(G)$ and let $x^{*C} = (x_e^*)_{e \in E(C)}$. For each cluster $C \in P$, let $\langle \tilde{x}^C, \tilde{f}^C \rangle$ be an optimal solution of CP(C). Then $g(\tilde{x}^C) \leq g(x^{*C})$.*

Proof. We argue that the vector $\langle x^{*C}, f^{*C} \rangle$, where $x_e^{*C} = x_e^*$ for all $e \in E(C)$ and $f_p^{*C} = f_p^*$ for all $p \in \mathcal{P}_{u,v}$, is a feasible solution to CP(C). By definition of $N(C)$ we have that for any $(u, v) \in N(C)$ all paths in $\mathcal{P}_{u,v}$ also appear in $G(C)$, and therefore $\langle x^{*C}, f^{*C} \rangle$ satisfies both capacity and flow constraints of CP(C) for pairs $(u, v) \in E(C)$ since they were satisfied in CP(G). Since we assumed that $\langle \tilde{x}^C, \tilde{f}^C \rangle$ is an optimal solution of CP(C), we get $g(\tilde{x}^C) \leq g(x^{*C})$. ◀

We now provide in Algorithm 2 a distributed algorithm for solving CP(G). The high level idea is the following: we partition the graph t times, have cluster centers solve CP(C) of their cluster using a sequential algorithm in each iteration, and then take an average over the solutions for each edge. Intuitively, for each edge, by averaging over local solutions for iterations in which the ball around that edge is in the same cluster, with high probability we get a feasible global solution. In the proof of Theorem 13, we will show that this solution gives a $(1 + \epsilon)$-approximation solution to the LP, for an arbitrary $0 < \epsilon \leq 1$.

We assume that all nodes know the values of D and ϵ. Let $C_{u,i}$ denote the cluster that node u belongs to in the i-th iteration, and let $\langle x^{C_{u,i},i}, f^{C_{u,i},i} \rangle$ be the fractional CP solution of $C_{u,i}$, where $\langle x_e^{C_{u,i},i}, f_p^{C_{u,i},i} \rangle$ is the fractional CP value for $e = (u, v)$, and $p \in \mathcal{P}_{u,v}$. Since the objective is a function of edge vectors, what we mean by having a distributed solution to a distance bounded network design CP is that each node u will know the value x_e for all the edges e incident to u. It is not hard to see that the algorithm could be modified so that every node u can also know the flow value f_p for each path p.

▶ **Theorem 13.** *Algorithm 2 takes $O((D/\epsilon)\log n)$ rounds to terminate, and it will compute a solution of cost at most $(1 + \epsilon)CP^*$ to a bounded distance network design CP (Definition 10) with high probability, where CP^* is the optimal solution and $0 < \epsilon \leq 1$. Moreover, if the convex program can be solved sequentially in polynomial time, then all of the node computations are also polynomial time.*

Proof. **Correctness:** We first show that with high probability the values $\tilde{x}_e, e \in E$ form a feasible solution. Here we only need to show that a feasible solution for the flow values exist, and do not require nodes to compute these values. Let $I_u = \{i : \exists C \in P_i, B(u,D) \subseteq C\}$, i.e. I_u is the set of iterations in which $B(u,D)$ is contained in a cluster, and let $I_{u,v}$ be the set of iterations in which both u and v are in the same cluster. Since we need to implement Algorithm 2 in a distributed manner, we use $I_{u,v}$ in our implementation, while the analysis is based on I_u. We can do so since by definition we have $I_u \subseteq I_{u,v}$, for any $(u,v) \in E$.

For any $p \in \mathcal{P}_{u,v}$, we set the flow values to be $\tilde{f}_p = \frac{1}{|I_u|}\sum_{i \in I_u} f_p^{C_{u,i},i}$,i.e. $\tilde{f}_p$ is the average over local flows in iterations in which $B(u,k)$ is fully contained in a cluster. We will show that this gives a feasible flow. First we argue that enough flow is being sent. For all $(u,v) \in \mathcal{S}$, we have,

$$\sum_{p \in \mathcal{P}_{u,v}} \tilde{f}_p = \sum_{p \in \mathcal{P}_{u,v}} \frac{1}{|I_u|}\sum_{i \in I_u} f_p^{C_{u,i},i} = \frac{1}{|I_u|}\sum_{i \in I_u}\sum_{p \in \mathcal{P}_{u,v}} f_p^{C_{u,i},i} \geq \frac{1}{|I_u|}\sum_{i \in I_u} 1 \geq 1.$$

We have used the fact that for each $i \in I_u$ the solution corresponding to the CP of the cluster containing u satisfies the constraint that $\sum_{p \in \mathcal{P}_{u,v}: e \in p} f_p^{C_{u,i},i} \geq 1$, because for each such i we know that $(u,v) \in N(C)$.

Next, we will argue that the capacity constraints are also satisfied. The second property of (D,λ)-padded decompositions implies that $\Pr(i \in I_u) \geq 1 - \lambda = 1 - \frac{\epsilon(1-\epsilon)}{(2-\epsilon)(1+\epsilon)} = \frac{1}{(1-\frac{\epsilon}{2})(1+\epsilon)}$ for each iteration $1 \leq i \leq t$. By linearity of expectations we have $E[|I_u|] \geq t(1-\lambda)$. Since each sampling is performed independently, by Chernoff bound for $\delta = \epsilon/2$, we get,

$$\begin{aligned}\Pr(|I_u| \leq t(1-\lambda)(1-\delta)) &= \Pr\left(|I_u| \leq \frac{t(1-\frac{\epsilon}{2})}{(1-\frac{\epsilon}{2})(1+\epsilon)}\right) \\ &= \Pr\left(|I_u| \leq \frac{t}{(1+\epsilon)}\right) \leq e^{-\frac{(\epsilon/2)^2(1-\lambda)t}{2}} \leq e^{-2\ln n} = \frac{1}{n^2}.\end{aligned}$$

Hence by a union bound on all nodes we have that with high probability $|I_u| > t/(1+\epsilon)$. Therefore, for all $(u,v) \in \mathcal{S}, e \in E$, we have (w.h.p.),

$$\begin{aligned}\sum_{p \in \mathcal{P}_{u,v}: e \in p} \tilde{f}_p &= \sum_{p \in \mathcal{P}_{u,v}: e \in p} \frac{1}{|I_u|}\sum_{i \in I_u} f_p^{C_{u,i},i} = \frac{1}{|I_u|}\sum_{i \in I_u}\sum_{p \in \mathcal{P}_{u,v}: e \in p} f_p^{C_{u,i},i} \leq \frac{1}{|I_u|}\sum_{i \in I_u} x_e^{C_{u,i},i} \\ &\leq \min\left(1, \frac{1}{|I_u|}\sum_{i \in I_{u,v}} x_e^{C_{u,i},i}\right) \leq \min\left(1, \frac{1+\epsilon}{t}\sum_{i \in I_{u,v}} x_e^{C_{u,i},i}\right) = \tilde{x}_e.\end{aligned}$$

Upper bound: We will now show that the upper bound holds. Let $\langle x^*, f^* \rangle$ be an optimal solution to CP(G). We have $\tilde{x}_e = \min(1, \frac{1+\epsilon}{t}\sum_{i \in I_e} x_e^{C_{u,i},i})$, and for each $e = (u,v)$ and $1 \leq i \leq t$, we set $\tilde{x}_e^i = x_e^{C_{u,i},i}$ if $i \in I_e$, and $\tilde{x}_e^i = 0$ otherwise. Note that $0 < (1+\epsilon)/t < 1$, and since g is a convex function and $g(\vec{0}) = 0$, by Jensen's inequality we have $g\left(\frac{1+\epsilon}{t}x\right) \leq \frac{1+\epsilon}{t}g(x)$. Then for $\tilde{x} = (\tilde{x}_e)_{e \in E}$ we can write:

$$\begin{aligned}g(\tilde{x}) = g\left((\tilde{x}_e)_{e \in E}\right) &\leq g\left(\frac{1+\epsilon}{t}\left(\sum_{i \in I_e} x_e^{C_{u,i},i}\right)_{e \in E}\right) \leq \frac{1+\epsilon}{t}g\left(\left(\sum_{i \in I_e} x_e^{C_{u,i},i}\right)_{e \in E}\right) \\ &\leq \frac{1+\epsilon}{t}g\left(\left(\sum_{i=1}^{t} \tilde{x}_e^i\right)_{e \in E}\right) \leq \frac{1+\epsilon}{t}g\left(\sum_{i=1}^{t}\left(\tilde{x}_e^i\right)_{e \in E}\right) \leq \frac{1+\epsilon}{t}\sum_{i=1}^{t} g\left(\left(\tilde{x}_e^i\right)_{e \in E}\right).\end{aligned}$$

In the final inequality, since g is convex, we used Jensen's inequality to take the sum out of the function. It is now enough to show that in each iteration i, it holds $g((\tilde{x}_e^i)_{e \in E}) \leq g(x^*)$.

Let $\tilde{x}^i = ((\tilde{x}^i_e)_{e \in E})$, and let $P_i = \{C_1, C_2, ..., C_\ell\}$ be the partition of V. Since g is a convex partitionable function w.r.t. G, there exists a nondecreasing and convex function $h : \mathbb{R}^m \mapsto \mathbb{R}$ for which we can write $g(\tilde{x}^i) = h_{P_i}(g(\tilde{x}^{i,C_1}), g(\tilde{x}^{i,C_2}), ..., g(\tilde{x}^{i,C_\ell}))$, since $\tilde{x}^i_e = 0$ by definition for edges which go between clusters (for simplicity we are denoting $\tilde{x}^{i^{C_j}}$ by $\tilde{x}^{i,C_j}$).

Recall that x^{*C} is the vector in which $x^{*C}_e = x^*_e$ for all $e \in E(C)$ and $x^{*C}_e = 0$ otherwise. By Lemma 12 we get that for all $C \in P_i$, $g(\tilde{x}^{i,C}) \leq g(x^{*C})$. Now we consider a vector $\hat{x}$, defined by setting $\hat{x}_e = x^*_e$ for all edge e with both endpoints in the same cluster, and $\hat{x}_e = 0$ otherwise. Since we assumed h_{P_i} to be nondecreasing, we get,

$$\begin{aligned} g(\tilde{x}^i) &= h_{P_i}(g(\tilde{x}^{i,C_1}), g(\tilde{x}^{i,C_2}), ..., g(\tilde{x}^{i,C\ell})) \leq h_{P_i}(g(x^{*C_1}), g(x^{*C_2}), ..., g(x^{*C_\ell})) \\ &= h_{P_i}(g(\hat{x}^{C_1}, \hat{x}^{C_2}, ..., g(\hat{x}^{C_\ell})) = g(\hat{x}) \leq g(x^*). \end{aligned}$$

For the last inequality we have used the fact that g is non-decreasing, and that for all $e \in E$, $\hat{x}_e \leq x^*_e$ (since either $\hat{x}_e = x^*_e$ or $\hat{x}_e = 0$). By plugging this into the above inequalities, we will get $g(\tilde{x}) \leq \frac{1+\epsilon}{t} \sum_{i=1}^{t} g(\tilde{x}^i) \leq (1+\epsilon) g(x^*)$, which implies the claim that Algorithm 2 gives a $(1+\epsilon)$-approximation to the optimal solution.

Time Complexity: The decomposition step and sending the information within a cluster takes $O((D/\epsilon)\log n)$ rounds since the diameter of each cluster is $O((D/\lambda)\log n) = O((D/\epsilon)\log n)$. Since each decomposition is independent, we can do all of them in parallel, so steps 1-4 of the algorithm only take $O((D/\epsilon)\log n)$ rounds in total. Clearly the rest of the algorithm can be done in a constant number of rounds. Hence in total w.h.p. the algorithm will take $O((D/\epsilon)\log n)$ rounds. ◀

5 Distributed Approximation Algorithms for Network Design

In this section, we will focus on several network design problems which can be approximated by first solving a convex relaxation using Algorithm 2 and then locally rounding the solution. For that purpose, we will describe how each problem has a distance bounded network design CP relaxation (Definition 10), and will then show that existing rounding schemes are local. All missing proofs can be found in Appendix C.

5.1 Directed k-Spanner

Dinitz and Krauthgamer [9] introduced a linear programming relaxation for Directed k-Spanner which is just a distance-bounded network design CP with demands pairs $\mathcal{S} = E$, allowed paths $\mathcal{P}_{u,v}$ which are the directed paths from u to v of length at most k, and objective function $g(x) = \sum_{e \in E} x_e$. They showed that this LP can be solved in polynomial time (approximately if k is non-constant). We will denote this LP by $LP(G)$. Clearly, LP(G) is a distance bounded network design CP with $D = k$. Hence, Theorem 13 implies that we can use Algorithm 2 to approximately solve this LP in $O(k \log n)$ rounds in the $\mathcal{LOCAL}$ model.

We now provide in Algorithm 3 a distributed rounding scheme that gives an $O(n^{1/2} \log n)$-approximation for Directed k-Spanner. This algorithm matches the best centralized approximation ratio known [5], and is just the obvious distributed version of the algorithm proposed in [5]. The difference is that here we truncate the shortest-path trees at depth k (as opposed to full shortest-path trees), and nodes choose whether to become a tree root independently (rather than chosen without replacement as in [5].

The following lemma is essentially from [5], with the proof requiring only slight technical changes due to the slightly different algorithms. We sketch it for completeness in Appendix C.

Algorithm 3: Distributed rounding algorithm for k-spanner.

Input : Graph $G = (V, E)$, fractional solution $\langle x, f \rangle$ to LP(G).

$E' = \emptyset, \forall v \in V : T_v^{in} = \emptyset, T_v^{out} = \emptyset$.

for $e \in E$ **do**

  Add e to E' with probability $\min(n^{1/2} \cdot \ln n \cdot x_e, 1)$.

for $v \in V$ **do**

  `// Random tree sampling`

  Choose p uniformly at random from $[0, 1]$.

  if $p < \frac{3 \ln n}{\sqrt{n}}$ **then**

    $T_v^{in} \leftarrow$ shortest path in-arborescence rooted at v truncated at depth k.

    $T_v^{out} \leftarrow$ shortest path out-arborescence rooted at v truncated at depth k.

Output $E' \cup (\cup_{v \in V}(T_v^{in} \cup T_v^{out}))$.`// A node knows its portion of the output.`

▶ **Lemma 14.** *Given a directed graph G, LP(G) as defined, and a fractional solution LP^* to LP(G), the output of Algorithm 3 has size $O(n^{1/2} \cdot (n + LP^*) \log n)$.*

It is easy to see that this algorithm can be implemented in the $\mathcal{LOCAL}$ model.

▶ **Lemma 15.** *Algorithm 3 runs in $O(k)$ time in the $\mathcal{LOCAL}$ model.*

We now immediately get our main result for Directed k-Spanner. Proof can be found in Appendix C.

▶ **Corollary 16.** *Algorithm 2 with $D = k$ along with the rounding scheme in Algorithm 3 yields an $O(n^{1/2} \ln n)$-approximation w.h.p. to* Directed k-Spanner *that runs in $O(k \log n)$ time in the $\mathcal{LOCAL}$ model and uses only polynomial-time computations at each node.*

5.2 Basic 3-Spanner and Basic 4-Spanner

If the input graph is undirected then stronger approximations are possible. In particular, for stretch 3 and 4, there are $\tilde{O}(n^{1/3})$-approximations due to [5] (for stretch 3) and [11] (for stretch 4). Without going into details, both of these algorithms use the same LP relaxation as in Directed k-Spanner, but round the LP differently. So in order to give distributed versions of these algorithms, we only need to modify Algorithm 3 to use the appropriate rounding algorithm (and change some of the other parameters in the shortest-path arborescence sampling). Fortunately, both of these algorithms use rounding schemes which are highly local. Informally, rather than sample each edge independently with probability proportional to the (inflated) fractional value as in Algorithm 3, these algorithms sample a value independently at each *vertex* and then include an edge if a particular function of the values of the two endpoints (different in each of the algorithms) passes some threshold. Clearly this is a very local rounding algorithm: once we have solved the LP relaxation using Theorem 13, each node can draw its random value and then spend one more round to exchange a message with each of its neighbors to find out their values, and thus determine which of the edges have been included by the rounding. Thus the total running time is dominated by the time needed to solve the LP, which in these cases is $O(\log n)$ using Theorem 13.

5.3 Lowest-Degree k-Spanner

We now turn our attention to Lowest-Degree k-Spanner: Given a graph $G = (V, E)$ and a value k, we want to find a k-spanner that minimizes the maximum degree. We will use the relaxation and rounding scheme proposed by Chlamtáč and Dinitz [6]. The linear programming relaxation used in [6] is very similar to the Directed and Basic k-spanner LP relaxation described earlier, with the difference being that a new variable λ is added to represent the maximum degree, and so the objective is to minimize λ and constraints are added to force λ to upper bound the maximum fractional degree. The proof of the following result can be found in Appendix C.

▶ **Theorem 17.** *Given a graph $G = (V, E)$ (directed or undirected), and any integer $k \geq 1$ there is a distributed algorithm that w.h.p. computes an $\tilde{O}(\Delta^{(1-1/k)^2})$-approximation to the* Lowest-Degree k-Spanner *problem, taking $O(k \log n)$ rounds of the $\mathcal{LOCAL}$ model and using only polynomial-time computations at each node.*

5.4 Directed Steiner Network with Distance Constraints

It is well-known that the centralized rounding of [5] for Directed k-Spanner is more general than is actually stated in their paper. In particular, the randomized rounding for "thin" edges gives the same guarantee even when each demand has a possibly different distance constraint. This fact was used, e.g., in [7] in their algorithms for Distance Preserver, Pairwise k-Spanner, and Directed Steiner Network with Distance Constraints. The difficulty in extending the algorithm of [5] is not in the LP rounding, but rather because the arborescence sampling technique used to handle thick edges in [5] (and in our Algorithm 3) assumes that n is a lower bound on the optimal cost. This assumption is true for Directed k-Spanner, but false for variants where there might be a tiny number of demands. However, it is easy to see that if we assume the demand graph is spanning (i.e., assume that every node is an endpoint of at least one demand) then the optimal solution must have at least $n/2$ edges, and hence we can again just use [5] to get a $\tilde{O}(\sqrt{n})$-approximation for Directed Steiner Network with Distance Constraints as long as the demand graph is spanning.

While this is in the centralized setting, since our algorithm for Directed k-Spanner is just a lightly modified distributed version of [5] (the only difficulty in the distributed setting is solving the LP, which is why that is the main technical contribution of this paper), we can easily modify it to give the same approximation for Directed Steiner Network with Distance Constraints with spanning demand graphs. The only change is that we use $D = \max_{(u,v)\in\mathcal{S}} L(u, v)$ instead of k when solving the linear programming relaxation (using Theorem 13) and when truncating the shortest-path arborescences that we sample (note that we have to assume that D is global knowledge, which is reasonable for spanner problems and for Shallow-Light Steiner Network but may be less reasonable for other special cases of Directed Steiner Network with Distance Constraints). This implies Theorem 4, and all of the interesting special cases (Shallow-Light Steiner Network, Distance Preserver, Pairwise k-Spanner, etc.) which it includes.

References

1. Amy Babay, Emily Wagner, Michael Dinitz, and Yair Amir. Timely, reliable, and cost-effective internet transport service using dissemination graphs. In *37th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 1–12, 2017.
2. Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theoretical Computer Science*, 2016.
3. Yair Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *FOCS'96*, pages 184–193, 1996.
4. Yair Bartal, John W. Byers, and Danny Raz. Global optimization using local information with applications to flow control. In *FOCS*, pages 303–312, 1997.
5. Piotr Berman, Arnab Bhattacharyya, Konstantin Makarychev, Sofya Raskhodnikova, and Grigory Yaroslavtsev. Improved approximation for the directed spanner problem. In *ICALP Part I*, pages 1–12, 2011.
6. Eden Chlamtác and Michael Dinitz. Lowest-degree k-spanner: Approximation and hardness. *Theory of Computing*, 12(1):1–29, 2016.
7. Eden Chlamtác, Michael Dinitz, Guy Kortsarz, and Bundit Laekhanukit. Approximating spanners and directed steiner forest: Upper and lower bounds. In *SODA*, 2017.
8. Michael Dinitz, Guy Kortsarz, and Ran Raz. Label cover instances with large girth and the hardness of approximating basic k-spanner. *ACM Trans. Algorithms*, 12(2):1–16, 2016.
9. Michael Dinitz and Robert Krauthgamer. Directed spanners via flow-based linear programs. In *STOC'11*, pages 323–332, 2011.
10. Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *PODC'11*, pages 169–178, 2011.
11. Michael Dinitz and Zeyu Zhang. Approximating low-stretch spanners. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, 2016.
12. Yevgeniy Dodis and Sanjeev Khanna. Design networks with bounded pairwise distance. In *STOC '99*, pages 750–759, 1999.
13. Michael Elkin. Personal Communication, 2017.
14. Michael Elkin and Ofer Neiman. Distributed strong diameter network decomposition: Extended abstract. In *PODC '16*, pages 211–216, 2016.
15. Patrik Floréen, Marja Hassinen, Joel Kaasinen, Petteri Kaski, Topi Musto, and Jukka Suomela. Local approximability of max-min and min-max linear programs. *Theory of Computing Systems*, 2011.
16. R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.
17. Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.
18. Anupam Gupta, Mohammad T. Hajiaghayi, and Harald Räcke. Oblivious network design. In *SODA '06*, pages 970–979, 2006.
19. M. Reza Khani and Mohammad R. Salavatipour. Improved approximations for buy-at-bulk and shallow-light k-steiner trees and (k,2)-subgraph. *Journal of Combinatorial Optimization*, 31(2):669–685, Feb 2016.
20. Robert Krauthgamer, James R. Lee, Manor Mendel, and Assaf Naor. Measured descent: A new embedding method for finite metrics. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS, pages 434–443, 2004.
21. Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *SODA '06*, pages 980–989, 2006.
22. Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, Dec 1993.

23 Christos H. Papadimitriou and Mihalis Yannakakis. Linear programming without the matrix. In *STOC '93*, pages 121–129, 1993.

24 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

25 David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

26 David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *PODC'87*, pages 77–85, 1987.

A Proofs from Section 3

A.1 Proof of Lemma 6

The first property in Definition 5 is directly implied by the definition of r_v for all nodes $v \in V$. For the second property we consider an arbitrary node $u \in V$, and compute the probability that the ball $B(u,k)$ is not in any of the clusters in P. Consider an arbitrary value $1 \leq t \leq n$, let $v \in V$ be the node such that $t = \pi(v)$, and let $z = z_v$ be the real number sampled by v. Also, for any $x, y \in V$, let $\tilde{d}(x,y) = \min(d(x,y), r \ln n + k)$. Let us also order the clusters based on their center's position in the permutation, so that C_t is the cluster corresponding to $t = \pi(v)$ (i.e. v is the cluster center of C_t). We define X_t to be the event that if $B(u,k)$ is not in the first $t-1$ clusters, then it is also not in any of the remaining clusters. We provide a recursive bound on X_t based on X_{t+1}. Then we will get the second property once we show $\Pr(X_0) \leq \epsilon$. We need to define the following events:

- A_t : $B(u,k)$ does not intersect with any of the clusters $C_1, .., C_{t-1}$.
- M_t^{cut} : $(\tilde{d}(v,u) - k \leq z < \tilde{d}(v,u) + k \mid A_t)$.
- M_t^{ex} : $(z < \tilde{d}(v,u) - k \mid A_t)$.
- X_t : $(\nexists j \geq t : B(u,k) \subseteq C_j \mid A_t)$.

In other words, conditional on the event that $B(u,k)$ is not in any of the first $t-1$ clusters, either $B(u,k) \subseteq C_t$, or else one the following two events will occur: M_t^{cut} is the event that $B(u,k)$ partially intersects C_t, and M^{ex} is the event that $B(u,k)$ does not intersect C_t.

Now the event X_t occurs only when either M_t^{cut} occurs or both M_t^{ex} and X_{t+1} occur (i.e. when $B(u,k)$ is not in C_t or any of the next clusters). Hence we can write $\Pr(X_t) \leq \Pr(M_t^{cut}) + \Pr(M_t^{ex}) \Pr(X_{t+1})$. Recall that z is independently sampled from the density function $p(z_v) = \left(\frac{n}{n-1}\right) \frac{e^{-z_v/r}}{r}$, and thus M^{cut} can be written as follows:

$$\begin{aligned}\Pr(M_t^{cut}) &= \int_{\tilde{d}(v,u)-k}^{\tilde{d}(v,u)+k} p(z)dz_z = \left(\frac{n}{n-1}\right)\left(1 - e^{-2k/r}\right) e^{-(\tilde{d}(v,u)-k)/r} \\ &\leq \left(\frac{n}{n-1}\right)\frac{2k}{r} e^{-(\tilde{d}(v,u)-k)/r}.\end{aligned}$$

Similarly, we can write,

$$\Pr(M_t^{ex}) = \int_0^{\tilde{d}(v,u)-k} p(z)dz_z = \left(\frac{n}{n-1}\right)\left(1 - e^{-(\tilde{d}(v,u)-k)/r}\right).$$

We now inductively prove that $\Pr(X_t) \leq (2 - \frac{t}{n-1})(\frac{2k}{r})$. If $t < n$ is the last step, then $\Pr(X_t) = 0$, and thus this bound clearly holds. Assume that the bound is true for X_{t+1}, we

show that then it also holds for X_t. We have,

$$\begin{aligned}\Pr(X_t) &\leq \Pr(M_t^{cut}) + \Pr(M_t^{ex})\Pr(X_{t+1}) \\ &\leq \left(\frac{n}{n-1}\right)\left(\frac{2k}{r}\right)\left(1+\frac{n-t-2}{n-1}\left(1-e^{-(\tilde{d}(v,u)-k)/r}\right)\right).\end{aligned}$$

Since $e^{-(\tilde{d}(v,u)-k)/r} \geq e^{-(\ln n)} \geq 1/n$, we get that $\Pr(X_t) \leq \left(2-\frac{t}{n-1}\right)\left(\frac{2k}{r}\right)$. The second property is then implied by the fact that $\Pr(X_0) \leq \frac{2k}{r} = \frac{2k}{2k(1/\epsilon)} = \epsilon$.

B Proofs from Section 4

B.1 Proof of Lemma 9

Let $\sigma = \{\sigma_1, ..., \sigma_\ell\}$ be a partition of nodes in V. For all $1 \leq i \leq \ell$, we have $g(x^{\sigma_i}) = \max_{v\in\sigma_i}(\sum_{u:(v,u)\in E} x^{\sigma_i}_{(v,u)})$. Then we can set $h_\sigma(y) = \max_{i\in[\ell]}(y_i), y \in \mathbb{R}^\ell$, where y_i is the i-th coordinate of y. Let $\sigma(v) \in \sigma$ be the cluster that node v belongs to. For all $x = (x_{(u,v)})_{(u,v)\in E}$, where $x_{(u,v)} = 0$ for any $(u,v) \in E$ s.t. $\sigma(u) \neq \sigma(v)$, we have,

$$\begin{aligned}g(x) &= \max_{v\in V}\left(\sum_{u:(v,u)\in E} x_{(v,u)}\right) = \max_{\sigma_i\in\sigma}\left(\max_{v\in\sigma_i}\left(\left(\sum_{u:(v,u)\in E} x^{\sigma_i}_{(v,u)}\right)\right)\right) \\ &= \max_{\sigma_i\in\sigma}\left(g\left(x^{\sigma_i}\right)\right) = h_\sigma(g(x^{\sigma_1}), g(x^{\sigma_2}), ..., g(x^{\sigma_\ell})).\end{aligned}$$

It is also easy to see that the function h_σ is convex and non-decreasing. Hence h_σ satisfies the conditions in Definition 8.

B.2 Proof of Lemma 11

The first property of (k,ϵ)-padded decompositions implies that for all nodes $u \in C$, we have $d(u,v) = O((k/\epsilon)\log n)$, where v is the center of cluster C. Each node $u \in C$ that determines v as the center of the cluster it belongs to, will send the information of its incident edges to v. Since there is no bound on the size of the messages being forwarded, this can be done in $O((k/\epsilon)\log n)$ time.

C Proofs from Section 5

C.1 Proof of Lemma 14

Let $N_{s,t}$ be the subgraph of G induced by the nodes on paths in $\mathcal{P}_{s,t}$. Edge $e \in E$ is called a *thick* edge if $|N_{s,t}| \geq n^{1/2}$, and otherwise it is called a *thin* edge. The set E' in Algorithm 3 satisfies the spanner property for all thin edges (as argued in [5]), and the random tree sampling phase satisfies the spanner property for the thick edges. Each thick edge (s,t) is spanned if at least one node in $N_{s,t}$ performs the random tree sampling. This probability is at least $1-(1-\frac{3\ln n}{n^{1/2}})^{n^{1/2}} \geq 1-1/n^3$. Then a union bound on all the edges (of size at most $O(n^2)$) implies that w.h.p. all thick edges are spanned. We now argue that the output is an $O(n^{1/2}\log n)$-approximation algorithm: at most $O(n^{1/2}\log n)$ arborescences are chosen with high probability (each arborscence has $O(n)$ edges), and we argued that $|E'| = O(n^{1/2}\log n \cdot LP^*)$. Hence, the overall size of the output is $O(n^{1/2}\log n \cdot (n + LP^*))$.

C.2 Proof of Lemma 15

Each node v in G has received the fractional solutions x_e corresponding to all edges $e \in E$ incident to v. The randomized rounding step can be performed locally: the node with the smaller ID flips a coin, and exchanges the coin flip result with its corresponding neighbors. In order to form T_i^{in} and T_i^{out}, v performs a distributed BFS algorithms by forming a shortest path tree while keeping track of the distance from v. When the distance counter reaches k, the tree construction terminates.

C.3 Proof of Corollary 16

We first run Algorithm 2 to solve LP(G) up to a constant factor (by setting $\epsilon = 1/2$), which takes time $O(k \log n)$ with high probability (Theorem 13). Since each cluster center can solve the local LP in polynomial time, all computations are polynomial time. We then use Algorithm 3 to round the fractional solutions of LP(G), which takes $O(k)$ time. Since the size of a k-spanner is at least $\Omega(n)$, Algorithm 3 then outputs an $O(n^{1/2} \ln n)$-approximation to the minimum (Lemma 14).

C.4 Proof of Theorem 17

It is easy to see that the LP relaxation proposed in [6] can be written as a distance bounded network design CP where the objective is $\max_{v \in V}(\deg(v)) = \max_{v \in V} \left(\sum_{u:\{v,u\} \in E} x_{\{v,u\}} \right)$ (we do not need to use their extra variable λ, since we can instead directly write the objective). Lemma 9 implies that this function is convex partitionable w.r.t. G, and hence the Lowest-Degree k-Spanner problem can be approximately solved (to within a constant factor) by using Algorithm 2 with $\epsilon = 1/2$. Next, we use the following rounding scheme proposed in [6]: each edge $e \in E$ is included in the spanner with probability $x_e^{1/k}$. It is clear that this can be done in a constant number of rounds, and hence the overall algorithm takes $O(k \log n)$ rounds (by Theorem 13) in the $\mathcal{LOCAL}$ model. In [6], it was shown that this leads to a $\tilde{O}(\Delta^{(1-1/k)^2})$-approximation solution of the problem.

Lower Bounds for Subgraph Detection in the CONGEST Model

Tzlil Gonen[1] **and Rotem Oshman**[2]

1 **Tel Aviv University, Tel Aviv, Israel**
`tzlilgon@mail.tau.ac.il`
2 **Tel Aviv University, Tel Aviv, Israel**
`roshman@mail.tau.ac.il`

Abstract

In the subgraph-freeness problem, we are given a constant-sized graph H, and wish to determine whether the network graph contains H as a subgraph or not. Until now, the only lower bounds on subgraph-freeness known for the CONGEST model were for cycles of length greater than 3; here we extend and generalize the cycle lower bound, and obtain polynomial lower bounds for subgraph-freeness in the CONGEST model for two classes of subgraphs.

The first class contains any graph obtained by starting from a 2-connected graph H for which we already know a lower bound, and replacing the vertices of H by arbitrary connected graphs. We show that the lower bound on H carries over to the new graph. The second class is constructed by starting from a cycle C_k of length $k \geq 4$, and constructing a graph $\tilde{H}$ from C_k by replacing each edge $\{i, (i+1) \bmod k\}$ of the cycle with a connected graph H_i, subject to some constraints on the graphs $H_0, \ldots, H_{k-1}$. In this case we obtain a polynomial lower bound for the new graph $\tilde{H}$, depending on the size of the shortest cycle in $\tilde{H}$ passing through the vertices of the original k-cycle.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity, F.2.0 General

Keywords and phrases subgraph freeness, CONGEST, lower bounds

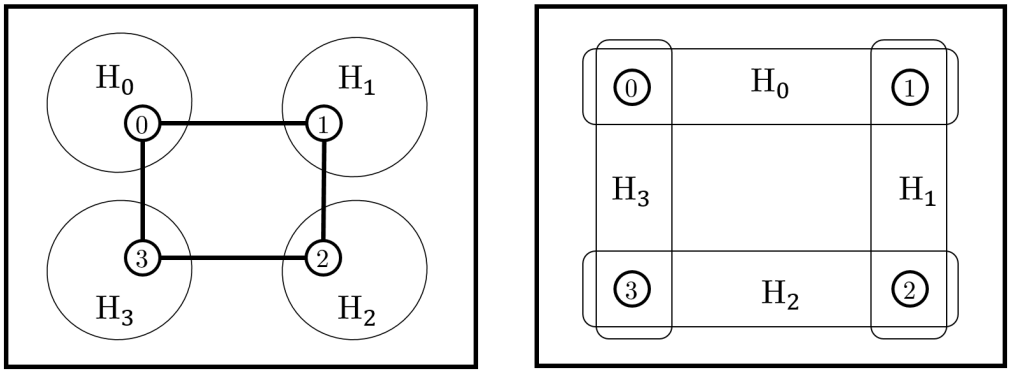
Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.6

1 Introduction

In the *subgraph-freeness* problem, we are given a constant-size graph H, and the goal is to determine whether the network graph contains a copy of H as a subgraph, or not. Subgraph-freeness in the CONGEST network model has recently received significant attention from the distributed computing community [4, 9, 10, 11, 12, 17], but until now, the only *lower bounds* in the literature have been for cycles: in [7] it was shown that checking C_k-freeness requires $\tilde{\Omega}(n)$ rounds for odd k and $\tilde{\Omega}(n^{2/k})$ rounds for even k. The lower bound for even cycles was recently strengthened to $\tilde{\Omega}(\sqrt{n})$ for any even k in [17]. In [1], it is shown that one-round deterministic algorithms for triangle-detection require bandwidth $\Omega(\Delta \log n)$, where Δ is the degree of the graph, and if we restrict to one bit per round, $\Omega(\log^* n)$ rounds are required.

In this work we seek to improve our understanding of the subgraph-freeness problem by broadening the class of graphs for which we know a polynomial lower bound, i.e., a lower bound of the form $\Omega(n^\delta)$ for some $\delta \in (0, 1]$. We give two classes of such graphs and prove polynomial lower bounds for them.

The first class of graphs comprises all graphs that can be constructed by starting from some 2-vertex-connected graph H for which we already *know* a polynomial lower bound



(a) Vertex replacement **(b)** Edge replacement

Figure 1

(regardless of how it was proven), and attaching an arbitrary graph to each vertex of H (see Fig. 1a). The graphs attached to different vertices may not share vertices or edges. We show that the new graph $\tilde{H}$ requires the same number of rounds as the graph H that we started with, up to a polylogarithmic factor, for both deterministic and randomized algorithms.

In particular, consider any connected graph G that is not a tree. (Trees are known to be easy: for any constant-sized tree T, the T-freeness problem can be deterministically solved in $O(1)$ rounds [8].) The *block cut tree* of G is a decomposition of G into a tree of maximal 2-connected components $C_1, \ldots, C_k$, where the pairwise intersection of any two components C_i, C_j (for $i \neq j$) is either empty or comprises a single vertex (called a *cut vertex*) [13]. Our first result shows that if for some i we know a lower bound of $\Omega(n^\delta)$ on checking C_i-freeness, then the entire graph G is also hard and requires $\tilde{\Omega}(n^\delta)$ rounds.

The second class of graphs takes a different approach: instead of replacing vertices, we replace edges (see Fig. 1b). We start with the 4-cycle[1] on $\{0, 1, 2, 3\}$, and replace each edge $\{i, (i+1) \bmod 4\}$ of the cycle with an arbitrary graph connecting vertices i and $(i+1) \bmod 4$, subject to two constraints: the resulting graph after replacing the edges must remain 2-connected, and no graph we added can contain the other three, connected to each other, as a subgraph (see Section 4 for a more formal definition). We show that checking subgraph-freeness for the resulting graph requires $\tilde{\Omega}(n^\delta)$ rounds, where the exponent $\delta \in (0, 1/2]$ depends on the graphs with which the edges of C_4 were replaced. For example, the second class of graphs includes any cycle of length at least 4 (for which our result is identical to [17]), as well as cycles with any number of chords, provided at least one of the smaller cycles created has length at least 4, and the smaller cycles are not too unbalanced in size.

The two classes complement each other: the graphs in the first class are obtained by starting from a 2-connected graph and replacing its vertices with other graphs, yielding a graph that is not 2-connected; the second class allows us to prove lower bounds on H-freeness for new subgraphs H that *are* 2-connected, and these can then be used to construct more graphs in the first class.

The reductions we use to show lower bounds for the two classes of graphs are fairly simple, but proving their correctness is non-trivial. For the first class, we take an algorithm for detecting the more complicated graph $\tilde{H}$, and transform it into an algorithm for the simpler graph H that we started with. This allows us to carry the lower bound over in the other

[1] We can also start from a larger cycle, but this gives us no additional power, because larger cycles can be constructed out of 4-cycles using our reduction. The lower bound we obtain would also not be higher.

direction. The transformation works by having each node in the network graph choose some vertex v in H, and "pretend" that it is the entire subgraph that we attached to vertex v in H when we constructed $\tilde{H}$. We must show that our algorithm does not inadvertently create copies of $\tilde{H}$ when the network graph does not contain a copy of H, and this is non-trivial. For the second class of graphs we extend the reduction from the two-party communication complexity of set disjointness used in [7] to show the hardness of C_4-freeness. Again, the difficulty lies in proving that in our reduction we do not create copies of $\tilde{H}$ when we do not mean to.

1.1 Related Work

The problem of subgraph-freeness (also called *excluded* or *forbidden subgraphs*) has been extensively studied in both the centralized and the distributed worlds. For the general problem of detecting whether a graph H is a subgraph of G, where both H and G are part of the input, the best known sequential algorithm is exponential [21]. When H is fixed and only G is the input, the problem becomes solvable in polynomial time. For example, using a technique called *color coding*, it is possible to detect a simple cycle of a specific size in expected time that is the running time of matrix multiplication [2]. We use the color coding technique in Section 3.

In the distributed setting, [10] and [11] very recently provided constant-round randomized and deterministic algorithms, respectively, for detecting a fixed tree in the CONGEST model. Both papers, as well as several others [3, 4, 12, 9] , also considered more general graphs, but with the exception of trees, they studied the *property testing* relaxation of the problem, where we only need to distinguish a graph that is H-free from a graph that is *far* from H-free. (In [9] there is also a property-testing algorithm for trees.) Here we consider the *exact* version.

Another recent work [15] gave randomized algorithms in the CONGEST model for triangle detection and triangle listing, with round complexity $\widetilde{O}(n^{2/3})$ and $\widetilde{O}(n^{3/4})$, respectively, and also established a lower bound of $\widetilde{\Omega}(n^{1/3})$ on the round complexity of triangle listing. There is also work on testing triangle-freeness in the congested clique model [5, 6] and in other, less directly related distributed models.

As for lower bounds on H-freeness in the CONGEST model, the only ones in the literature (to our knowledge) are for cycles. (In [7] there are lower bounds for other graphs, in a broadcast variant of the CONGEST model where nodes are required to send the *same* message on all their edges.) For any fixed $k > 3$, there is a polynomial lower bound for detecting the k-cycle C_k in the CONGEST model: it was first presented by [7], which showed that $\Omega(\mathrm{ex}(n, C_k)/\log(n))$ rounds are required, where $\mathrm{ex}(n, C_k)$ is the largest possible number of edges in a C_k-free graph over n vertices (see [14] for a survey on extremal graphs with forbidden subgraphs). In particular, for odd-length cycles, the lower bound of [7] is nearly linear. Very recently, [17] improved the lower bound for even-length cycles to $\Omega(\sqrt{n}/\log(n)))$.

2 Preliminaries

We generally work with undirected graphs, unless indicated otherwise.

The CONGEST model is a synchronous network model, where computation proceeds in *rounds*. In each round, each node of the network may send B bits on each of its edges, and these messages are received by neighbors in the current round. As is typical in the literature, we assume $B = O(\log n)$ here (the lower bounds generalize to other settings of B in a straightforward manner).

Notation. We let $V(G), E(G)$ denote the vertex and edge set of graph G, respectively, and use the following short-hand notation:

- $H \subseteq G$ for two graphs H, G stands for the subgraph relation: $H \subseteq G$ iff $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.
- We use $A \hookrightarrow B$ to denote a function mapping all elements of A into B. Specifically, if G contains a copy of H as a subgraph, we frequently let $\sigma : H \hookrightarrow G$ denote an isomorphism mapping the nodes of H into $V(G)$. When the isomorphism is *onto*, we use the usual $\rightarrow$, i.e., $\sigma : H \rightarrow H'$.
- If $\sigma : H \hookrightarrow G$ is an isomorphism mapping H onto a copy of H in G, we let $\sigma(H)$ denote the image of H under σ. We have $\sigma(H) \subseteq G$.

▶ **Definition 1** (Subgraph freeness)**.** Fix a graph H of constant size. In the *H-freeness* problem, the goal is to determine whether the input graph G contains a copy of H as a subgraph or not, that is, whether there is a subgraph $G' \subseteq G$ which is isomorphic to H.

We say that a distributed algorithm A *solves H-freeness with success probability p* if

- When A is executed in a graph containing a copy of H, the probability that all nodes accept is at least p.
- When A is executed in an H-free graph, the probability that at least one node rejects is at least p.

We typically assume constant p, e.g., $p = 2/3$.

The graph classes we define assume some amount of *vertex connectivity*, defined below.

▶ **Definition 2** (Vertex connectivity)**.** We say that a graph G is k-connected, for $k \geq 1$, if removing any k vertices from G (along with all their incident edges) does not disconnect G.

In Section 4 we rely on a lower bound for *two-party communication complexity*: we have two players, Alice and Bob, with private inputs X, Y, respectively. The players wish to compute a joint function $f(X, Y)$ of their inputs, and we are interested in the total number of bits they must exchange to do so (see the textbook [18] for more background on communication complexity).

In particular, we are interested in the *set disjointness* function, where the inputs X, Y are interpreted as subsets $X, Y \subseteq [n]$, and the goal of the players is to determine whether $X \cap Y = \emptyset$. The celebrated lower bound of [16, 20] shows that even for randomized communication protocols, the players must exchange $\Omega(n)$ bits to solve set disjointness with constant success probability.

3 Vertex-Replacement Reduction

In this section we describe a reduction that allows us to take any 2-connected graph H for which we know a polynomial lower bound, attach arbitrary connected graphs to the vertices of H to obtain a new graph $\tilde{H}$, and obtain the same lower bound on $\tilde{H}$ as for H, up to a logarithmic factor.

Let us describe more formally what we mean by "attaching graphs to the vertices of H".

▶ **Definition 3** (Graph attachment)**.** Fix two graphs G, H over vertex sets $V(G), V(H)$ with a unique intersection, $V(G) \cap V(H) = \{v\}$. We define an *attached graph*, $G \cup H$, as follows: $V(G \cup H) = V(G) \cup V(H)$, and $E(G \cup H) = E(G) \cup E(H)$. We refer to vertex v as the *attachment point* of $G \cup H$.

The attachment operation trivially has the following property, which we rely on in the sequel:

▶ **Property 4.** *Let $G = H_1 \cup H_2$, with $v \in V(H_1) \cap V(H_2)$ the attachment point. Then for any two vertices $w_1 \in V(H_1), w_2 \in V(H_2)$, any path $u_1 = w_1, u_2, \ldots, u_\ell = w_2$ from w_1 to w_2 in H must include v (that is, for some $1 \leq i \leq \ell$ we have $u_i = v$).*

▶ **Definition 5** (Compatible graphs)**.** Let G be a graph with $V(G) = [s]$. We say that a sequence of graphs $H_0, \ldots, H_{s-1}$ is *compatible with* G if

(1) For any $i \in [s]$ we have $V(H_i) \cap V(G) = \{i\}$, and

(2) For any $i \neq j$ we have $V(H_i) \cap V(H_j) = \emptyset$.

If $H_0, \ldots, H_{s-1}$ are compatible with G, we use the short-hand notation $G \cup \bigcup_{i=0}^{s-1} H_i$ to denote the graph defined by attaching each H_i to G (the order in which we attach $H_0, \ldots, H_{s-1}$ does not matter).

▶ **Definition 6** (The class $\mathcal{A}_H$)**.** Fix a graph H on vertices $V(H) = [s]$. The class $\mathcal{A}_H$ includes any graph given by $\tilde{H} = H \cup \bigcup_{i=0}^{s-1} H_i$ for $H_0, \ldots, H_{s-1}$ compatible with H.

See Figure 1a for an illustration.

Now we can state our main theorem for this section.

▶ **Theorem 7.** *Let H be any 2-connected graph on $V(H) = [s]$. If solving H-freeness requires $\Omega(n^\delta)$ rounds in graphs of size n for deterministic algorithms, then for any $\tilde{H} \in \mathcal{A}_H$, solving $\tilde{H}$-freeness requires $\Omega(n^\delta / \log n)$ rounds for deterministic algorithms. The same relationship holds for randomized algorithms, except that if checking H-freeness requires $\Omega(n^\delta)$ rounds for randomized algorithms, then checking $\tilde{H}$-freeness requires $\Omega(n^\delta / \log^2 n)$ rounds.*

To prove Theorem 7, we describe a *randomized* reduction, which takes an algorithm for checking $\tilde{H}$-freeness, and constructs a randomized algorithm for checking H-freeness. This allows us to carry the lower bound over in the other direction, from checking H-freeness to checking $\tilde{H}$-freeness. This prove Theorem 7 for randomized algorithms; to prove it for deterministic algorithms, we derandomize our reduction.[2]

3.1 The Reduction

Fix an algorithm $\tilde{A}$ for checking $\tilde{H}$-freeness, with running time $t(n)$ and success probability p. We wish to use $\tilde{A}$ to check H-freeness. To do this, we have each node v of the network choose a "role in H", $c(v) \in V(H)$; we call $c(v)$ the *color* of v. Each node v then "imagines" that it is attached to a copy of $H_{c(v)}$, and we simulate the run of $\tilde{A}$ in the resulting network.

In the randomized reduction, each node chooses a *random* color; we call a copy of H *properly colored* if each node chose a color that matches the vertex it is mapped to in H.

▶ **Definition 8** (Properly-colored copies)**.** Fix graphs G, H, such that G contains a copy of H. Let $\sigma : H \hookrightarrow G$ map H onto its copy in G. We say that $\sigma(H)$ is *properly colored* by an assignment of colors $c : V(G) \to V(H)$ if for each $v \in \sigma(H)$ we have $c(v) = \sigma^{-1}(v)$.

Next we formally describe the algorithm $A(c)$ that is executed for a given color assignment $c : V(G) \to V(H)$, where G is the network graph.

[2] It is also possible to first derandomize the reduction and then apply it to either a randomized or deterministic algorithm. However, we must first reduce the error probability of the randomized algorithm, so either way we lose an additional $\log n$ factor.

3.1.1 Construction of $A(c)$

We define a "virtual graph", $\tilde{G}_c$, where

(1) Each vertex $v \in V(G)$ is replaced by $(v, c(v))$; let G_c be the resulting copy of G.

(2) Each $v \in V(G)$ creates a "virtual copy" $\tilde{G}_v$ of the graph $H_{c(v)}$, where each vertex $i \in H_{c(v)}$ is replaced by (v, i).

(3) $\tilde{G}_c = G_c \cup \bigcup_{v \in V(G)} \tilde{G}_v$, that is, $\tilde{G}_c$ is obtained by attaching the copies $\tilde{G}_v$ for each $v \in V(G)$ to G_c.

Let $\tilde{U}_v = V(\tilde{G}_v)$ denote the "virtual vertices" corresponding to the copy of $H_{c(v)}$ attached to vertex v (i.e., to vertex (v, c) in $\tilde{G}_c$).

In $A(c)$, we *simulate* the execution of $\tilde{A}$ in the virtual graph $\tilde{G}_c$, with each vertex $v \in G$ simulating all the vertices in $\tilde{U}_v$ (including "itself", vertex $(v, c(v))$. At the end of the execution, each vertex v *accepts* if all virtual vertices in $\tilde{U}_v$ accepted, and otherwise it *rejects*.

The running time of $A(c)$ in G is the same as the running time of $\tilde{A}$ in $\tilde{G}_c$. Let $a = \max_i |V(H_i)|$ be the maximum number of vertices simulated by a vertex of G in $\tilde{G}_c$. Then $|V(\tilde{G}_c)| \leq a|V(G)|$. Thus, the worst-case running time of A in G is bounded by $\tilde{t}(an)$.

The algorithm A itself simply has each node v choose a random color $c(v) \in V(H)$, and then runs $A(c)$. We will show that

- If the graph contains a copy of H, then A accepts with probability at least p/s^s;
- If the graph does not contain a copy of H, then A accepts with probability at most $1 - p$.

3.1.2 What Could Go Wrong?

Before proving A correct, let us illustrate why the requirement that H be 2-connected is necessary, and in general what could go wrong if we are not careful.

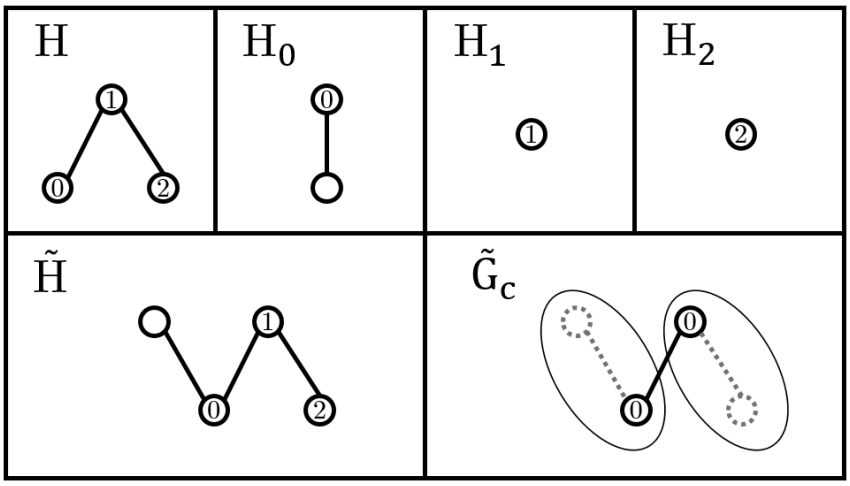


Figure 2 A bad example with a subgraph H that is not 2-connected.

Consider the graph H which is the line over vertices $\{0, 1, 2\}$, in increasing order. H is of course *not* 2-connected. Let H_0 be the graph containing only a single edge, $\{0, a\}$, and for each $i = 1, 2$, let H_i be the single vertex i. The graph $\tilde{H}$ we obtain by attaching H_0, H_1, H_2 to H is the line of length 3 (i.e., 3 edges).

Now let G be a graph comprising a single edge, $\{a, b\}$, and suppose that we were unlucky and chose the color assignment $c(a) = c(b) = 0$. Then $\tilde{G}_c$ is a line of length 3, i.e., it contains

$\tilde{H}$ as a subgraph. But G does *not* contain H as a subgraph, so our reduction would be unsound if we applied it with this H.

Next we show that for subgraphs H that *are* 2-connected, our reduction is complete and sound: the graph we construct contains a copy of $\tilde{H}$ iff the original graph contained a properly-colored copy of H.

3.1.3 Completeness of A

Suppose that the original graph G contains a copy H, and let $\sigma : H \hookrightarrow G$ be an isomorphism mapping H into G. Suppose further that $\sigma(H)$ is properly colored by c. Conditioned on this event, the new graph $\tilde{G}_c$ contains a copy of $\tilde{H}$, witnessed by the following isomorphism σ_c: for each $x \in \tilde{H}$, if $x \in H_j$, then $\sigma_c(x) = (\sigma(j), x)$. Thus, conditioned on the event that we get a good coloring c, $A(c)$ accepts with probability at least p. The overall acceptance probability of A is at least p/s^s.

3.1.4 Soundness of A

Now suppose that G does *not* contain a copy of H; we show that for any coloring c of G, the corresponding graph $\tilde{G}_c$ does not contain a copy of $\tilde{H}$. Therefore A rejects with the same probability that $\tilde{A}$ rejects, i.e., at least p.

Suppose for the sake of contradiction that there is a coloring c of G such that $\tilde{G}_c$ does contain a copy of $\tilde{H}$, and let us abuse notation by denoting $\tilde{G} = \tilde{G}_c$. We will show that there is some "virtual copy" $\tilde{G}_v$ which contains infinitely many distinct copies of H, which is, of course, impossible.

First we show that any copy of H in $\tilde{G}$ must be entirely contained in some $\tilde{G}_v$.

▶ **Observation 9.** *For any isomorphism $\sigma : H \hookrightarrow \tilde{G}$, there is a vertex $v \in V(G)$ such that $\sigma(H) \subseteq \tilde{U}_v$.*

Proof. From our assumption, there is no copy of H in G, and therefore $\sigma(H) \not\subseteq G$; that is, $\sigma(H)$ must contain some "virtual nodes". Accordingly, there must be some "virtual part", $\tilde{U}_v$, such that $\sigma(H) \cap \tilde{U}_v \setminus \{v\} \neq \emptyset$. If $\sigma(H)$ is only *partially* contained in $\tilde{U}_v$ (i.e., if $\sigma(H) \not\subseteq \tilde{U}_v$), then removing $(v, c(v))$ from $\tilde{G}$ disconnects $\sigma(H)$, because any path from vertices in $\tilde{U}_v$ to vertices outside $\tilde{U}_v$ must pass through $(v, c(v))$ (Property 4). But H is 2-connected, and therefore $\sigma(H)$ must be entirely contained in $\tilde{U}_v$. ◀

Now fix an isomorphism $\sigma : \tilde{H} \hookrightarrow \tilde{G}$, and let v be the vertex from Observation 9, such that $\sigma(H) \subseteq \tilde{U}_v$. Let $i = c(v)$.

▶ **Observation 10.** $\sigma(H_i) \not\subseteq \tilde{G}_v$.

Proof. Recall that $\tilde{U}_v$ is the vertex set of $\tilde{G}_v$, which is a copy of H_i; thus, $|\tilde{U}_v| = |V(H_i)|$, and we can only have $\sigma(H_i) \subseteq \tilde{U}_v$ if $\sigma(H_i) = \tilde{U}_v$. But $\tilde{U}_v$ contains at least one vertex which is not in $\sigma(H_i)$: take any vertex $j \in V(H) \setminus \{i\}$ (which exists because $|V(H)| > 1$), and since $\sigma(H) \subseteq \tilde{U}_v$ and $V(H) \cap V(H_i) = \{i\}$, we have $\sigma(j) \in \tilde{U}_v \setminus \sigma(H_i)$. Therefore $\sigma(H_i) \neq \tilde{U}_v$ and $\sigma(H_i) \not\subseteq \tilde{U}_v$. ◀

▶ **Observation 11.** *We have $(v, i) \in \sigma(H_i)$.*

Proof. From Observation 10, there is some $u \in H_i$ such that $\sigma(u) \notin \tilde{U}_v$. In H_i, there is a path π from u to i, because $u, i \in H_i$ and H_i was assumed connected; the isomorphism σ maps π onto a path $\sigma(\pi)$ in $\tilde{G}$ from $\sigma(u) \notin \tilde{U}_v$ to $\sigma(i) \in \sigma(H) \subseteq \tilde{U}_v$. By Property 4, the path $\sigma(\pi)$ must include (v, i), and therefore $(v, i) \in \sigma(H_i)$. ◀

For a vertex $(v,x) \in \tilde{G}_v$, let σ' be the isomorphism between $\tilde{G}_v$ and H_i given by $\sigma'(v,x) = \sigma(x)$. Note that $\sigma'(\tilde{G}_v) \subseteq \sigma(H_i) \subseteq \tilde{G}_v$.

We now construct a sequence of isomorphisms $\sigma_1, \sigma_2, \ldots : H \to \tilde{G}_v$ as follows:

- $\sigma_0 = \sigma$,
- $\sigma_{j+1}(x) = \sigma'(\sigma_j(x))$ for any $j \geq 0, x \in H$.

Note that a-priori, this sequence is not necessarily well-defined, because σ' can map nodes of $\tilde{G}_v$ to nodes outside $\tilde{G}_v$. We will show that this does not happen to nodes of H, so that we get infinitely many copies of H inside $\tilde{G}_v$.

For any $j > 0$, let

$$d_j = \mathrm{dist}_{\tilde{G}_v}((v,i), \sigma_j(i)).$$

Also, let π_j be a path of length d_j between (v,i) and $\sigma_j(i)$ in $\tilde{G}_v$. (These definitions assume that the sequence $\sigma_1, \ldots, \sigma_j$ is well-defined up to index j, and accordingly we will only use them under this assumption.)

▶ **Observation 12.** *Let $\pi \subseteq \tilde{G}_v$ be a simple path between two vertices $(v,x),(v,y) \in \tilde{G}_v$, such that $\sigma'(v,x), \sigma'(v,y) \in \tilde{G}_v$ as well. Then $\sigma'(\pi) \subseteq \tilde{G}_v$.*

Proof. Assume for contradiction that $\sigma'(\pi) \not\subseteq \tilde{G}_v$, and let $(v,w) \in \pi$ be some vertex such that $\sigma'(v,w) \notin \tilde{G}_v$. Split π into two sub-paths, π_1 and π_2, where π_1 connects (v,x) to (v,w), and π_2 connects (v,w) to (v,y). The isomorphism σ' maps π_1, π_2 into two simple paths $\sigma'(\pi_1), \sigma'(\pi_2)$ connecting $\sigma'(v,x)$ to $\sigma'(v,w)$ and $\sigma'(v,w)$ to $\sigma'(v,y)$, respectively. Since $\sigma'(v,w) \notin \tilde{G}_v$ and $\sigma'(v,x), \sigma'(v,y) \in \tilde{G}_v$, Property 4 asserts that $\sigma(\pi_1)$ and $\sigma(\pi_2)$ both include node (v,i). But this means that $\sigma(\pi) = \sigma(\pi_1)\sigma(\pi_2)$ is not a simple path, which is a contradiction, as π is simple and σ' is bijective. ◀

For convenience, let us denote $\sigma_0(i) = (v,i)$.

▶ **Claim 13.** *Fix $k > 0$, and assume that $\sigma_j(H) \subseteq \tilde{G}_v$ for each $j < k$. Then for any $0 \leq j < k$ and $0 < \ell < k$ with $j + \ell < k$, there is a simple path $\pi \subseteq \tilde{G}_v$ of length d_ℓ connecting $\sigma_j(i)$ and $\sigma_{j+\ell}(i)$.*

Proof. By induction on j.

For $j = 0$, this is immediate from the definition of d_ℓ as the distance in $\tilde{G}_v$ between $\sigma_0(i) = (v,i)$ and $\sigma_\ell(i)$.

For the induction step, suppose that the claim holds for j: there is a simple path $\pi \subseteq \tilde{G}_v$ of length d_ℓ connecting $\sigma_j(i)$ and $\sigma_{j+\ell}(i)$. Assume that $j + 1 + \ell < k$, and recall that we assumed $\sigma_r(H) \subseteq \tilde{G}_v$ for each $0 \leq r < k$. Then in particular, $\sigma_{j+1}(i) = \sigma'(\sigma_j(i)) \in \tilde{G}_v$ and $\sigma_{j+\ell+1}(i) = \sigma'(\sigma_{j+\ell}(i)) \in \tilde{G}_v$. Thus we can apply Observation 12 to get that $\sigma'(\pi) \subseteq \tilde{G}_v$. This proves the claim, because $\sigma'(\pi)$ connects $\sigma_{j+1}(i)$ and $\sigma_{j+\ell+1}(i)$ and has length d_ℓ. ◀

▶ **Corollary 14.** *Fix $k > 0$, and assume that $\sigma_j(i) \in \tilde{G}_v$ for each $j < k$. Then for any $0 \leq j < k$ and $0 < \ell < k$ with $j + \ell + 1 \leq k$, there is a simple path of length d_ℓ connecting $\sigma_j(i)$ and $\sigma_{j+\ell}(i)$.*

Proof. If $j + \ell + 1 < k$, this is Claim 13. If $j = 0$, this follows from the definition of d_j, as in the base case of Claim 13. Finally, if $j + \ell + 1 = k$, then Claim 13 shows that there is a path of length d_ℓ between $\sigma_{j-1}(i) \in \tilde{G}_v$ and $\sigma_{j+\ell-1}(i) \in \tilde{G}_v$, and applying σ' once more yields a path of length d_ℓ between $\sigma'(\sigma_{j-1}(i)) = \sigma_j(i)$ and $\sigma'(\sigma_{j+\ell-1}(i)) = \sigma_{j+\ell}(i)$. ◀

▶ **Claim 15.** *For each $k \geq 0$ we have $\sigma_k(H) \subseteq \tilde{G}_v$.*

Proof. By induction on k. The base case, $k = 0$, is by choice of v, i.

For the induction step, let $k \geq 1$, and assume that $\sigma_j(H) \subseteq \tilde{G}_v$ for each $j < k$.

It suffices to find some $j < k$ such that there is a path of length at most d_j between $\sigma_j(i)$ and $\sigma_k(i)$: since d_j is the distance from $\sigma_j(i)$ to (v, i), any path of length at most d_j starting at $\sigma_j(i)$ ends inside $\tilde{G}_v$. We therefore get that $\sigma_k(i) \in \tilde{G}_v$, and since any copy of H is entirely contained in some $\tilde{G}_w$ for $w \in V(G)$ (Observation 9), $\sigma_k(H) \subseteq \tilde{G}_v$.

Let us write $k = 2j + b$ for $j > 0$ and $b \in \{0, 1\}$. By Corollary 14, there is a path $\pi \subseteq \tilde{G}_v$ of length d_j connecting $\sigma_{j+b}(i)$ and $\sigma_{2j+b}(i) = \sigma_k(i)$. If k is even, i.e., $b = 0$, then we are done, as there is a path of length d_j between $\sigma_j(i)$ and $\sigma_k(i)$.

Suppose now that k is odd, i.e., $b = 1$. Then we just showed that there is a path of length d_j connecting $\sigma_{j+1}(i)$ and $\sigma_k(i)$. If $d_j \leq d_{j+1}$, then we are done. Otherwise, $d_j > d_{j+1}$. By Corollary 14, there is a path of length $d_{j+1} < d_j$ between $\sigma_j(i)$ and $\sigma_{2j+1}(i) = \sigma_k(i)$, so again we are done. ◀

Next we show that the copies $\sigma_1(H), \sigma_2(H), \ldots$ cannot overlap completely.

▶ **Claim 16.** *For each $1 \leq j < k$ we have $\sigma_j(H) \neq \sigma_k(H)$.*

Proof. By induction on j.

For the base case, observe that for any $k > 1$ we have $\sigma(H) \neq \sigma_k(H)$: since $H \cap H_i = \{i\}$, we know that $\sigma(H) \cap \sigma(H_i) = \{\sigma(i)\}$, but for all $k > 1$ we have $\sigma_k(H) \subseteq \sigma(H_i)$. Therefore $\sigma(H) \cap \sigma_k(H) \subseteq \{\sigma(i)\}$, and since $|H| > 1$ we get that $\sigma(H) \neq \sigma_k(H)$.

Now suppose the claim holds for j, and suppose for the sake of contradiction that $\sigma_{j+1}(H) = \sigma_k(H)$ for some $k > j+1$. Since $\sigma_{j+1}(H) = \sigma'(\sigma_j(H))$ and $\sigma_k(H) = \sigma'(\sigma_{k-1}(H))$, and σ' is bijective, we get that $\sigma_j(H) = \sigma_{k-1}(H)$, which contradicts the induction hypothesis (as $k > j - 1$). ◀

We have reached a contradiction: we have an infinite sequence of copies $\sigma_1(H), \sigma_2(H), \ldots$ all contained in $\tilde{G}_v$ but no two are identical. This is impossible, because $\tilde{G}_v$ is finite.

3.2 Lower Bound for Randomized Algorithms

We can now use the reduction described above to prove Theorem 7 for randomized algorithms.

Proof of Theorem 7 for randomized algorithms. Fix an algorithm $\tilde{A}$ for solving $\tilde{H}$-freeness, with success probability $p > 1/2$ and running time $t(n)$.

Our first step is to construct from $\tilde{A}$ an algorithm $\tilde{A}'$, such that in graphs of size $s \cdot n$, the success probability of $\tilde{A}'$ in solving $\tilde{H}$-freeness is at least $p' \geq p$ such that $1 - p' < p'/s^s$, that is, $p' > 1/(1 + 1/s^s)$. To do this, we execute $\tilde{A}$ sequentially $C \cdot \log n$ times for some constant C, and have each vertex accept iff the majority of iterations of $\tilde{A}$ accepted. If the graph is not $\tilde{H}$-free, the expected number of executions that accept is at least $Cp \log n$, and if the graph is $\tilde{H}$-free, the expected number of iterations that accept is at most $C(1 - p) \log n$. We choose C a sufficiently large constant so that by Chernoff, if the graph contains a copy of $\tilde{H}$, then a given vertex rejects with probability at most $p'/(sn)$, and if the graph is $\tilde{H}$-free, then the probability that all vertices accept is at most p'. A union bound then gives the desired success probability for $\tilde{A}'$.

Next, we use the transformation described above to obtain an algorithm A for solving H-freeness with running time $(Ct(sn) \log(n))$ in graphs of size sn. In graphs that contain a copy of H, the probability that A accepts is at least p'/s^s, and in graphs that are H-free the probability that A accepts is at most $1 - p'$. Since we chose p' so that $1 - p' < p'/s^s$, we can again use $C' \cdot \log n$ iterations of A for some constant C' to increase the success probability to $2/3$. The resulting running time is $C \cdot C' \cdot t(sn) \log^2 n$, and we know that $\Omega(n^\delta)$ rounds are required for solving H-freeness; therefore, $t(N) = \Omega(N^\delta / \log^2 N)$. ◀

3.3 Derandomizing the Reduction

To obtain a deterministic reduction, we use a trick often used in the context of graph coloring. We fix a set of L color assignments $c_1, \ldots, c_L : [n] \to [s]$ with the following property:[3]

▶ **Property 17.** *Take any graph G on n vertices containing a copy of H, and let $\sigma : H \hookrightarrow G$ map H onto its copy in G. Then $\sigma(H)$ is properly colored by at least one color assignment c_i for $i \in [L]$.*

▶ **Lemma 18.** *There is a list $c_1, \ldots, c_L$ with $L = O(\log n)$ satisfying Property 17.*

Proof. We choose L random color assignments, and show that for $L = O(\log n)$, the probability that the list we sampled satisfies Property 17 is greater than zero.

Let $s = |V(H)|$, and let $L = \alpha \log n$, where $\alpha \geq 1$ is a constant we will fix later. Consider a specific copy $\sigma(H)$ of H, identified by the vertices $v_1, \ldots, v_s \in [n]$. The probability that a random assignment $c : V(G) \to V(H)$ colors $\sigma(H)$ properly is $1/s^s$. The probability that c *fails* to color $\sigma(H)$ properly is $1 - 1/s^s$, and the probability that L random assignments all fail to color $\sigma(H)$ properly is

$$\left(1 - \frac{1}{s^s}\right)^L \leq \left(e^{-1/s^s}\right)^L = e^{-L/s^s} = \left(\frac{1}{n}\right)^{\alpha/s^s}.$$

By union bound, the probability that a list of L random color assignments is bad for *any* copy of H is bounded by $n^s \cdot (1/n)^{\alpha/s^s}$, and choosing α a sufficiently large constant, this probability is strictly smaller than one. ◀

▶ **Corollary 19.** *If solving H-freeness requires $\Omega(n^\delta)$ rounds in graphs of size n for deterministic algorithms, then solving $\tilde{H}$-freeness also requires $\Omega(n^\delta / \log n)$ rounds for deterministic algorithms.*

Proof. Fix $c_1, \ldots, c_L : V(G) \to V(H)$ satisfying Property 17, with $L = O(\log n)$.

Given a deterministic algorithm $\tilde{A}$ for testing $\tilde{H}$-freeness, we construct a deterministic algorithm A for testing H-freeness, by repeating the simulation from the proof of Theorem 7 with each c_i for $i \in [L]$. If at least one iteration of $\tilde{A}$ accepts, then A accepts.

If $\tilde{A}$ requires $t(n)$ rounds in graphs of size n, then A requires $t(sn) \log n$ rounds.

When we run A in a graph that does not contain a copy of H, we showed that *no* color assignment produces a copy of $\tilde{H}$, and therefore $\tilde{A}$ rejects in all of its iterations. Therefore, A rejects as well.

When we run A in a graph that contains a copy of H, there is some i such that c_i colors the copy of H properly, and in this case we showed that the virtual graph $\tilde{G}$ contains a copy of $\tilde{H}$. In iteration i, $\tilde{A}$ accepts, and therefore so does A. ◀

4 Edge-Replacement Reduction

In this section we describe another reduction, which replaces the *edges* of a graph with other graphs. We choose to work with the 4-cycle C_4 as our starting graph whose edges we replace. (Any larger cycle would do just as well but would not give us any additional results, as larger cycles can be constructed from C_4 by replacing a single edge with a line comprising several edges.)

[3] Here we assume that the nodes of the graph in which our algorithm is executed have IDs $\{1, \ldots, n\}$, but this assumption is not necessary. The IDs can be drawn from a polynomially-large namespace; the proof of Lemma 18 still goes through, with a larger constant.

▶ **Definition 20** (The class $\mathcal{B}$)**.** A graph H on vertices $V = \{0, \ldots, k-1\}$ is in $\mathcal{B}$ if it satisfies the following conditions:

(1) H is 2-connected, and

(2) There are four subsets $V_0, \ldots, V_3 \subseteq V$ such that

- **(a)** $V_i \cap V_{(i+1) \bmod 4} = \{(i+1) \bmod 4\}$ for each $i \in \{0, \ldots, 3\}$,
- **(b)** $V_i \cap V_j = \emptyset$ for each $i, j \in \{0, \ldots, 3\}$ where $j \neq (i+1) \bmod 4$,
- **(c)** The subgraph H_i induced by H on V_i is connected,
- **(d)** For each $i \in [3]$, the graph H_i does not contain a copy of the other three attached to each other, $\bigcup_{j \neq i} H_j$.

For an illustration, see Figure 1b. We believe the last requirement is not necessary for the reduction, but our current proof of soundness requires it.

Our approach for proving a lower bound for graphs in the class $\mathcal{B}$ is to modify the reduction from [7], which was originally used to show a lower bound of $\tilde{\Omega}(\sqrt{n})$ for checking C_4-freeness. As in [7], we require dense graphs that are free of cycles up to some length. We use the construction of [19], as stated in [14], Theorem 4.47:

▶ **Theorem 21** ([19])**.** *For any $n, g \geq 1$ there is a bipartite graph on n vertices, with girth at least $2g+2$, and $\Omega(n^{1+\epsilon(g)})$ edges, where*

$$\epsilon(g) = \begin{cases} 2/(3g-3), & \text{if } g \text{ is odd,} \\ 2/(3g-3+1), & \text{if } g \text{ is even.} \end{cases}$$

Now we can state our result:

▶ **Theorem 22.** *For each $H \in \mathcal{B}$ there is some $\delta \in (0, 1/2]$ such that testing H-freeness requires $\Omega(n^{\delta})$ rounds for randomized algorithms.*

The proof is a reduction from two-party communication complexity: we show that if there is a fast algorithm for H-freeness for some $H \in \mathcal{B}$, then we can construct from this algorithm a communication-efficient protocol for set disjointness. Because we know that set disjointness requires $\Omega(n)$ bits of communication, we obtain a lower bound on the number of rounds required for H-freeness for $H \in \mathcal{B}$.

Proof of Theorem 22. For each $i = 0, \ldots, 3$, let k_i be the distance between i and $(i+1) \bmod 4$ in H_i, and let $k = \sum_{i=0}^{3} k_i$. Assume w.l.o.g. that $k_0 + k_2 \geq k_1 + k_3$ and $k_2 \geq k_0$. Observe that H contains the cycle C_k, passing through nodes $0, 1, 2, 3$ in this order.

Fix a bipartite graph F with girth greater than $\lfloor k/k_0 \rfloor$ and with m edges, $\{e_1, \ldots, e_m\}$. Direct the edges from one side to the other, and let us denote $(u, v)^{-1} = (v, u)$.

We reduce from set disjointness on m elements as follows.

Given inputs $X, Y \subseteq [m]$, Alice and Bob jointly construct a graph $G_{X,Y}$, which consists of many copies of $H_0, \ldots, H_3$, where

- Alice decides which copies of H_0 to include in $G_{X,Y}$, depending on her input X;
- Bob decides which copies of H_1 to include in $G_{X,Y}$, depending on his input Y;
- The set of copies of H_1, H_3 that are included in $G_{X,Y}$ is fixed and does not depend on the input.

The various copies of $H_0, \ldots, H_3$ are connected to each other at vertices $0, \ldots, 3$ in such a way that a copy of H appears iff $X \cap Y \neq \emptyset$.

Each of the copies of H_i is associated with a directed edge $e \in [n]^2$, and accordingly, we name each copy H_i^e, where i is the index of the graph H_i of which H_i^e is a copy, and $e \in [n]^2$ is the edge with which it is associated.

We now describe the construction formally.

Step 1: Creating copies of $H_0, \ldots, H_3$

For an edge $e = (u, v) \in [n]^2$ and a graph H_i where $i \in [3]$, we create a *copy of* H_i denoted $H_i^{(u,v)}$, over the vertices

$$V\left(H_i^{(u,v)}\right) = \{(i, u), ((i+1) \bmod 4, v)\} \cup \{(x, i, (u, v)) \mid x \in V(H_i) \setminus \{i, (i+1) \bmod 4\}\}.$$

The original H_i is mapped onto its copy $H_i^{(u,v)}$ by the isomorphism $\sigma_i^{(u,v)}$, defined as follows:

- The vertices i and $(i+1) \bmod 4$ of H_i are mapped by $\sigma_i^{(u,v)}$ to $(i, u), ((i+1) \bmod 4, v)$ in $H_i^{(u,v)}$, respectively. Vertices y are called the *endpoints* of $H_i^{(u,v)}$. Note that these vertices can be *shared* between different copies $H_i^{e_1}$ and $H_j^{e_2}$; we elaborate below, in Property 23.
- Any other vertex, $x \in V(H_i) \setminus \{i, (i+1) \bmod 4\}$, is mapped by $\sigma_i^{(u,v)}$ to a "fresh vertex" denoted $(x, i, (u, v))$, which is not shared with any other copy.

As we said, some of the copies share vertices; the following property characterizes the copies that do and do not share vertices, and which vertices are shared.

▶ **Property 23.** *For any* $(i, u_i, v_i) \neq (j, u_j, v_j)$,

- *If* $j = (i+1) \bmod 4$ *and* $v_i = u_j$, *then* $V(H_i^{(u_i,v_i)}) \cap V(H_j^{(u_j,v_j)}) = \{((i+1) \bmod 4, v_i)\}$;
- *If* $i = (j+1) \bmod 4$ *and* $v_j = u_i$, *then* $V(H_i^{(u_i,v_i)}) \cap V(H_j^{(u_j,v_j)}) = \{((j+1) \bmod 4, v_j)\}$;
- *If neither condition above holds, then* $V(H_i^{(u_i,v_i)}) \cap V(H_j^{(u_j,v_j)}) = \emptyset$.

Selecting copies for $G_{X,Y}$

Not all the copies H_i^e described above actually appear in the graph $G_{X,Y}$ constructed by the players; as we said, the players choose which copies to include based on their inputs.

Recall that F is a bipartite graph with girth greater than $\lfloor k/k_0 \rfloor$ and with m edges, $\{e_1, \ldots, e_m\}$, which we directed from one side to the other. Recall also that $X, Y \subseteq [m]$, so Alice and Bob can view their inputs as sets of edges from F. The graph $G_{X,Y}$ is constructed by attaching together four sets of copies:

$$G_{X,Y} = \bigcup_{i=0}^{3} \left\{ H_i^e \mid e \in E_i^{X,Y} \right\},$$

where the edge set $E_i^{X,Y}$ is given by:

- For $i \in \{1, 3\}$ we define $E_i^{X,Y} = \{(u, u) \mid u \in [n]\}$,
- For $i = 0$ we define $E_0^{X,Y} = \{e_x \mid x \in X\}$, and
- For $i = 2$ we define $E_2^{X,Y} = \left\{e_y^{-1} \mid y \in Y\right\}$.

The Simulation

We describe how Alice and Bob simulate the execution of a distributed algorithm for testing H-freeness in the graph $G_{X,Y}$. Let G_A be the subgraph of $G_{X,Y}$ induced by all the nodes participating in copies H_0^e for some $e \in F$, and let G_B be the subgraph of $G_{X,Y}$ induced by all the nodes participating in copies $H_2^{e^{-1}}$ for some $e \in F$. Alice simulates all the nodes in G_A, and Bob simulates the nodes in G_B; the remaining nodes are simulated by both players, using public randomness to agree on their random choices. Note that by construction, each player knows all internal edges on their side of the graph (G_A, G_B respectively), and both players know the structure of the shared part of the graph.

In each round, Alice sends Bob the messages sent on each edge adjacent to nodes $(0, u)$, $(1, u)$ for each $u \in [n]$, and Bob sends Alice the messages sent on each edge adjacent to nodes $(2, u)$, $(3, u)$ for each $u \in [n]$. The players then feed these messages to the nodes they simulate, and also compute the messages sent on internal edges in G_A (for Alice) or G_B (for Bob), and feed them in as well.

If Δ is the total degree of nodes $0, 1, 2, 3$ in H, then the total number of bits required for each round of the simulation is $\Delta n \log n = O(n \log n)$.

Correctness of the Reduction

First, suppose $x \in X \cap Y$, and let $e_x = (u, v)$. Then Alice and Bob both add copies $H_0^{(u,v)}, H_2^{(v,u)}$, with $H_0^{(u,v)}$ connecting nodes $(0, u)$ and $(1, v)$, and $H_2^{(v,u)}$ connecting nodes $(2, v)$ and $(3, u)$. Because in $G_{X,Y}$ there is always a copy $H_1^{(v,v)}$ of H_1 connecting $(1, v)$ to $(2, v)$, and there is always a copy $H_3^{(u,u)}$ of H_3 connecting $(3, u)$ to $(0, u)$, we get a complete copy of H.

Now suppose that $X \cap Y = \emptyset$, and assume for the sake of contradiction that $G_{X,Y}$ contains a copy of H.

▶ **Observation 24.** *In $G_{X,Y}$ there is no copy of H which intersects at most one copy $H_i^{(u,v)}$ for each $i \in [3]$.*

Proof. Suppose there is such a copy. Because $|V(H)| = \sum_{i=0}^{3} |V_i|$, this copy of H must also intersect at *least* one copy $H_i^{(u_i,v_i)}$ for each $i \in [3]$, and since H is connected, these four subgraphs must share vertices (as there are no edges between the non-shared vertices of two distinct copies). By Property 23, we must have $v_i = u_{(i+1) \bmod 4}$ for each $i \in [3]$. In particular, $v_0 = u_1 = v_1 = u_2$, and $v_2 = u_3 = v_3 = u_0$. But such copies would only be added to the graph if there exist edges $e_x = (u_0, v_0) \in E(F)$ with $x \in X$, and $e_y = (v_2, u_2) \in E(F)$ with $y \in Y$. Since $v_2 = u_0$ and $u_2 = v_0$, we get that $x = y$ and hence $X \cap Y \neq \emptyset$, contrary to our assumption. ◀

▶ **Claim 25.** *In $G_{X,Y}$, any cycle C_d with $d \leq k$ is entirely contained in some copy H_i^e for $i \in [3]$ and $e \in [n]^2$.*

Proof. Consider a cycle C_d in $G_{X,Y}$ which is not entirely contained in any copy H_i^x. Let $(i_1, e_1), \ldots, (i_a, e_a)$ be the sequence of copies of $H_{i_j}^{e_j}$ through which C_d passes. Then $C' = e_1, \ldots, e_a$ is a cycle: the various copies connect to each other only at their endpoints, and if C_d enters some copy through one endpoint it must exit at the other (it is a simple cycle, so it cannot exit through the vertex where it entered).

Recall that F is a bipartite graph with girth greater than $\lfloor k/k_0 \rfloor$.

Consider first a cycle C_d completely contained in Alice's side of the graph, G_A. Then each edge e_j of C' corresponds to a passage through copy $H_0^{e_j}$ and adds at least k_0 edges, as this is the distance between vertices 0 and 1 inside H_0. However, the girth of F is greater than $\lfloor k/k_0 \rfloor$, so $|C'| > \lfloor k/k_0 \rfloor$, that is, C_d must pass through at least $\lfloor k/k_0 \rfloor + 1$ copies before the cycle can close. This yields a total length of at least $(\lfloor k/k_0 \rfloor + 1) \cdot k_0 > k \geq d$.

Next, suppose that C_d is entirely contained in G_B. The same argument holds, except that each passage through a copy of H_2 adds k_2 edges instead of k_0. Since $k_2 \geq k_0$ by assumption, we still get a total length of at least $(\lfloor k/k_0 \rfloor + 1) \cdot k_2 \geq (\lfloor k/k_0 \rfloor + 1) \cdot k_0 > k \geq d$.

Finally, suppose C_d is not entirely contained in G_A or in G_B. Because the copies $\left\{H_1^{(u,u)}, H_3^{(u,u)}\right\}_{u \in [n]}$ are not connected to each other, and C_d is not contained in any single

copy, it must go through some copies of H_0 and H_2 as well. We know from Observation 24 that C_d *cannot* use only one copy of H_i for each $i \in [3]$.

Since F is bipartite, C_d must use at least two copies of H_0 from Alice's side, two copies of H_2 from Bob's side, and either two copies of H_1 or two copies of H_3. Therefore the length of the cycle is at least $2(k_0 + k_2 + \min(k_1, k_3)) \geq k + 2\min(k_1, k_3) > k \geq d$. ◀

▶ **Corollary 26.** *There exist $i \in [3]$ and $u, v \in [n]$ such that for each $j \neq i, j \in [3]$ we have $\sigma(H_j) \subseteq H_i^{(u,v)}$.*

Proof. From the definition of the class $\mathcal{B}$, the graph H contains a cycle C_k passing through vertices $0, 1, 2, 3$, and from Observation 25, σ maps this cycle into some copy, that is, $\sigma(C_k) \subseteq H_i^{(u,v)}$ for some $i \in [3]$ and $u, v \in [n]$.

Recall that $H_i^{(u,v)}$ is connected to the rest of $G_{X,Y}$ only at its own endpoints (i, u) and $((i+1) \bmod 4, v)$, and that in H, each subgraph H_j connects to the next subgraph $H_{(j+1) \bmod 4}$ at the endpoint $(j+1) \bmod 4$. This means that there is at most one $j \in [3]$ such that $\sigma(H_j) \not\subseteq H_i^{(u,v)}$: all paths from $\sigma(0), \ldots, \sigma(3)$ to vertices outside $H_i^{(u,v)}$ must pass through an endpoint of $H_i^{(u,v)}$, either (i, u) or $((i+1) \bmod 4, v)$. Thus, if there is some $x \in H_j$ such that $\sigma(x) \notin H_i^{(u,v)}$, then any path from $\sigma(x)$ to $\sigma(j)$ must pass through (i, u) or through $((i+1) \bmod 4, v)$. In fact, because H is 2-connected, there must be *two* paths from $\sigma(x)$ to $\sigma(j)$, one using (i, u) and one using $((i+1) \bmod 4, v)$, otherwise removing one endpoint would disconnect H. But this implies that $(i, u), ((i+1) \bmod 4, v) \in \sigma(H_j)$, and hence for any $j' \neq j$ there is no path inside $\sigma(H_{j'})$ from $\sigma(j')$ to any node outside $H_i^{(u,v)}$ Because $H_{j'}$ is connected, we get that $\sigma(H_{j'}) \subseteq H_i^{(u,v)}$.

We have now shown that there is at most one $j \in [3]$ such that $\sigma(H_j) \not\subseteq H_i^{(u,v)}$. Note in addition that we cannot have $\sigma(H_i) \subseteq H_i^{(u,v)}$, as $H_i^{(u,v)}$ is isomorphic to H_i, and in addition at least two nodes in $H_i^{(u,v)}$ are not in the σ-image of H_i (nodes $(i+2) \bmod 4$ and $(i+3) \bmod 4$). It follows that if some H_j "escapes" $H_i^{(u,v)}$, i.e., if $\sigma(H_j) \not\subseteq H_i^{(u,v)}$, then $j = i$. In other words, for each $j \neq i$ we have $\sigma(H_j) \subseteq H_i^{(u,v)}$. ◀

From the corollary we get that there is some $i \in [3]$ which contains a copy of $\bigcup_{j \neq i} H_j$, contradicting condition (d) of the class $\mathcal{B}$. This concludes the proof of soundness for the reduction.

Putting Everything Together

Let $g = \lfloor k/k_0 \rfloor$. By Theorem 21, there is a bipartite graph with girth greater than g and $m = \Omega(n^{1+\epsilon})$ edges, where $\epsilon = \Theta(1/g)$. Set F to be such a graph.

The size of the graph $G_{X,Y}$ is bounded by $m \cdot (|V_0| + |V2|) + n \cdot (|V_1| + |V_3|) \leq n^{1+\epsilon} \cdot s$, where $s = |V(H)|$. We know that to solve disjointness over m elements we require a total of $\Omega(m)$ bits, and our simulation requires $O(n \log n)$ bits per round, so the number of rounds of any randomized algorithm testing H-freeness in $G_{X,Y}$ is $\Omega(m/(n \log n)) = \Omega(n^{\epsilon}/\log n)$ in the worst case.

Now let us express this result in terms of the size of the graph $G_{X,Y}$: set $N = |V(G_{X,Y})| \leq n^{1+\epsilon}s$, where s is the size of H (a constant). Then $n = \Theta(N^{1/(1+\epsilon)})$, and the running time we get is $\Omega(N^{\epsilon/(1+\epsilon)}/\log N)$ rounds. ◀

5 Conclusion

In this paper we made a step towards understanding which subgraphs are hard to detect in the distributed setting: we showed that if H is a 2-connected graph that is hard to detect, then any graph obtained from H by replacing its vertices with other graphs will also be hard; and that if we take a cycle and replace its edges with other graphs, then under some conditions, then resulting graph will still be hard.

In our view, the main open question raised by our work is the following: *is there a 2-connected graph H such that H-freeness can be solved in sub-polynomial time? Or are all 2-connected graphs hard?*

As we pointed out in Section 1, if indeed all 2-connected graphs are polynomially hard, then our first reduction implies that *any graph that is not a tree is polynomially hard*, yielding a strong dichotomy between trees, which only require $O(1)$ rounds to detect [8], and any other connected graph.

References

1 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Christoph Lenzen. Fooling views: A new lower bound technique for distributed computations under congestion. *CoRR*, abs/1711.01623, 2017.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

3 Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. *Distributed Computing*, 24(2):79–89, 2011.

4 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. *Fast Distributed Algorithms for Testing Graph Properties*, pages 43–56. Springer, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-53426-7_4.

5 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 143–152, 2015.

6 Danny Dolev, Christoph Lenzen, and Shir Peled. *"Tri, Tri Again": Finding Triangles and Small Subgraphs in a Distributed Setting*, pages 195–209. Springer, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-33651-5_14.

7 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 367–376, 2014.

8 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three Notes on Distributed Property Testing. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:30, 2017.

9 Guy Even, Reut Levi, and Moti Medina. Faster and simpler distributed algorithms for testing and correcting graph properties in the congest-model. *CoRR*, abs/1705.04898, 2017.

10 Orr Fischer, Tzlil Gonen, and Rotem Oshman. Distributed property testing for subgraph-freeness revisited. *CoRR*, abs/1705.04033, 2017.

11 Pierre Fraigniaud, Pedro Montealegre, Dennis Olivetti, Ivan Rapaport, and Ioan Todinca. Distributed subgraph detection. *CoRR*, abs/1706.03996, 2017.

12 Pierre Fraigniaud, Ivan Rapaport, Ville Salo, and Ioan Todinca. *Distributed Testing of Excluded Subgraphs*, pages 342–356. Springer, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-53426-7_25.

13 Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, 1985.

14 Zoltán Füredi and Miklós Simonovits. *The History of Degenerate (Bipartite) Extremal Graph Problems*, pages 169–264. Springer, Berlin, Heidelberg, 2013.

15 Taisuke Izumi and François Le Gall. Triangle finding and listing in congest networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 381–389, 2017.

16 Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discrete Math.*, 5(4):545–557, 1992.

17 Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST. *CoRR*, abs/1705.10195, 2017.

18 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.

19 Felix Lazebnik, Vasiliy A. Ustimenko, and Andrew J. Woldar. A new series of dense graphs of high girth. *Bull. Amer. Math. Soc.*, 32(1):73–39, 1995.

20 Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992.

21 J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

Extending Transactional Memory with Atomic Deferral

Tingzhe Zhou[1], Victor Luchangco[2], and Michael Spear[3]

1 **Lehigh University, Bethlehem, USA**
`tiz214@lehigh.edu`
2 **Oracle Labs, Burlington, USA**
`victor.luchangco@oracle.com`
3 **Lehigh University, Bethlehem, USA**
`spear@lehigh.edu`

Abstract

This paper introduces *atomic deferral*, an extension to TM that allows programmers to move long-running or irrevocable operations out of a transaction while maintaining serializability: the transaction and its deferred operation appear to execute atomically from the perspective of other transactions. Thus, programmers can adapt lock-based programs to exploit TM with relatively little effort and without sacrificing scalability by atomically deferring the problematic operations. We demonstrate this with several use cases for atomic deferral, as well as an in-depth analysis of its use on the PARSEC dedup benchmark, where we show that atomic deferral enables TM to be competitive with well-designed lock-based code.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Transactional Memory, Concurrency, Synchronization, I/O

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.7

1 Introduction

Transactional memory (TM) [9], originally proposed as a hardware extension to facilitate the creation of scalable nonblocking data structures, is beginning to be widely available. It is supported by major hardware vendors [14], and a Technical Specification for C++ Extensions for Transactional Memory (henceforth, TMTS) has been proposed [10], and is (partially) implemented by the GCC compiler [5], with support for both hardware (HTM) and software (STM) implementations. Thus, programmers are at last able to use TM in production.

The appeal of TM is its simplicity: a programmer need only wrap an operation inside a language-level "transaction"; a run-time system executes the transaction using custom hardware and/or compiler-generated software instrumentation. The run-time system monitors the low-level memory accesses of transactions, and allows concurrent transactions to execute simultaneously as long as their memory accesses do not conflict. TM is particularly appealing for data structures and applications with irregular or hard-to-predict memory accesses (e.g., the rebalancing operations of a red-black tree mutation), making them difficult to implement efficiently using locks. As long as conflicts are rare, so that transactions can run concurrently with few rollbacks, and the implementation of TM itself does not introduce too much overhead, a TM-based implementation should be comparable in performance to lock-based code when running single-threaded, and scale much better with multiple threads.

Unfortunately, there are only a few examples of TM being used in "real" software [11,18,25]. Why is TM not more widely adopted? One reason is that TM implementations often do

introduce significant overhead, especially when transactions are large and there is no hardware support. Another is that adapting a program to use TM is not always a simple matter of replacing lock-based critical sections with transactions because transactions cannot execute some kinds of operations (e.g., I/O and certain system calls), and most TMs do not support condition synchronization and other important synchronization patterns. Achieving good performance with TM often requires significant changes to the code both to reduce the size and number of large transactions and to move "unsafe" operations within critical sections out of transactions while still ensuring correct synchronization.

In this paper, we introduce language and run-time support for the *atomic deferral* of operations in transactions: deferred operations do not execute until after the transaction commits. Unlike prior work on deferred operations, atomic deferral does not violate serializability: concurrent transactions cannot observe an intermediate state in which the transaction's updates are complete but its deferred operation's updates are not. This property is particularly important for operations that perform output: if the output fails, compensating or retrying operations can be performed as part of the deferred operation so that it appears to be atomic with the deferring transaction.

We implement atomic deferral by introducing *transaction-friendly locks*, that is, locks that can be acquired and released within transactions, and to which transactions can "subscribe". With these locks, programmers can "mix and match" lock-based and transaction-based synchronization, using whichever is appropriate to the need. We use these locks to protect shared data accessed by deferred operations. The atomicity of the transaction and its deferred operation is preserved by acquiring the appropriate locks before committing the transaction.

The contributions of this paper are as follows:

- We introduce a transaction-friendly implementation of mutual exclusion locks.
- We present a mechanism for atomically deferring complex operations in transactions.
- We describe a compiler extension that allows the compiler to coordinate deferred operations with concurrent transactions without additional programmer effort.
- We describe use cases for atomic deferral in benchmarks and real-world programs.
- We show how atomic deferral can eliminate scalability bottlenecks in microbenchmarks and the PARSEC dedup workload.

The remainder of this paper is organized as follows: In Section 2, we review salient features of modern TM implementations. We provide an example of atomic deferral behavior in Section 3. Section 4 describe atomic deferral, and how it can be implemented using transaction-friendly locks. We discuss use cases in Section 5, and evaluate our implementation in Section 6. We describe related work and future research directions in Sections 7 and 8.

2 Background: Transactional Memory

In this section, we describe salient features of TM, particularly as it is specified in the C++ TMTS [10] and implemented by GCC. The TMTS specifies two lexical blocks for using TM: *atomic blocks* and *synchronized blocks*. Code within atomic blocks is restricted to ensure that the effects of an aborted transaction can be efficiently undone. To enforce this restriction statically, functions called while executing an atomic block must be declared *transaction-safe*. Synchronized blocks lift this restriction at the cost of possibly serializing all transactions (from both atomic and synchronized blocks) when executing an irrevocable operation (i.e., an operation that cannot be undone).

Although TM does not support condition synchronization, the *retry* operation proposed by Harris et al. [6] can produce a similar effect: A transaction that finds that some required

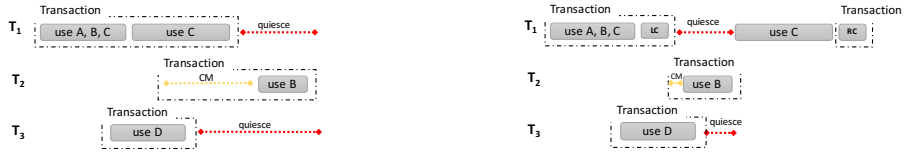


Figure 1 Motivation for atomic defer. On the left, T_1's transaction includes a long running operation using C. On the right, C is locked, and then the operation on C is deferred until after the transaction commits. The use of locking and deferral of the operation on C enables the operations by threads T_2 and T_3 to progress more quickly, without violating serializability.

condition does not hold can invoke retry, which aborts the transaction and does not reschedule it until some location in its read set has changed. Thus, the transaction appears to execute only from a state in which the desired condition holds. (This differs from traditional condition synchronization in that the transaction aborts, and so its effects before discovering that the condition does not hold are undone.) The TMTS does not support retry directly, but it does provide the ability for atomic blocks to abort themselves, which we can use to approximate retry (albeit inefficiently).

The TMTS does not separate transactional and nontransactional memory: any location can be accessed both transactionally and nontransactionally. This presents a challenge to STM implementations known as the *privatization problem* [19]: Although semantically concurrent transactional and nontransactional access to the same location is a data race (which has undefined semantics in C++), a thread that uses a transaction to remove an object from a shared data structure and accesses it nontransactionally afterwards may conflict with another transaction that accessed the same object concurrently but is still "cleaning up". To avoid this problem, a thread must not access a privatized object nontransactionally until every transaction that may have accessed that object has completed entirely (i.e., committed or aborted, including clean up). Since an STM cannot, in general, determine when an object is privatized, implementations typically wait after committing any writing transaction until every concurrent transaction has completed entirely; this waiting is called *quiescing*.

When transactions repeatedly fail to commit due to repeated conflicts with other threads, a TM implementation may manage contention by delaying some transactions to increase the likelihood that others complete. Although contention management policies vary, most TM implementations employ serialization as a last resort: a transaction that fails too many times will request that all other transactions abort, and no new transactions commence, until it completes. Unless the workload exhibits pathological conflicts, serialization should be rare. In GCC, the default is for software transactions to serialize after 100 attempts, and hardware transactions to serialize after 2. Dynamically tuning this parameter has been shown to have a significant impact on some workloads [4]. Any nontrivial amount of serialization, however, has a terrible effect on performance, particularly because serialization delays all active transactions, even those from completely unrelated parts of the program (unlike locked critical sections, which are partitioned by the locks that they acquire).

3 A Motivating Example

To motivate atomic deferral, consider the execution depicted on the left side of Figure 1, which captures behavior we observed when transactionalizing the PARSEC dedup kernel [1]. T_1 executes a transaction that first accesses locations A, B and C, and then does a lengthy operation that accesses only C. Concurrently, T_2 executes a transaction that accesses B.

Because these transactions conflict, either one of them must abort, or T_2 must wait until T_1 commits before it can proceed with the part of its transaction that accesses B, which is what happens in this case. Because T_1 may privatize some memory, after it commits, it must quiesce, waiting for T_2 to finish (either commit or abort).

The situation is even worse for T_3, which accesses a completely different location, and so does not conflict with either T_1's or T_2's transaction. Nonetheless, T_3 might privatize some memory, and thus it must quiesce until all concurrent transactions complete, so it must wait for T_1's lengthy operation to complete, and then for T_2's transaction to complete afterwards , before it can proceed.

Note, however, that T_1 is only accessing C in the lengthy operation at the end of its transaction. If it could defer that operation until after it commits, then T_2 could start the section of its code that accesses B earlier, and likely commit before T_1 completes its lengthy operation on C. T_3 can also stop quiescing earlier (i.e., when T_2 commits). This case is depicted one the right side of Figure 1.

One problem with doing this, however, is that a thread accessing C after T_1 commits the initial part of its transaction but before T_1 finishes its final lengthy operation on C will see an intermediate state of T_1's transaction, violating atomicity. To avoid that, we should prevent other threads from accessing C in that interval. This is represented by the small "LC" and "RC" operations (for "lock C" and "release C" respectively). Achieving this is the core conceptual contribution of this paper, and we show how to do it in the next section.

Prior studies of concurrent applications [12, 18, 21, 23, 24] found that output operations and long-running operations occur often while locks are held. The consequences of such operations are less severe in lock-based code than in programs with TM, primarily because the lock-based programs use many locks: a long-running operation protected by lock L_1 does not impede a thread executing a critical section protected by L_2. However, long-running operations tend to hold as few locks as necessary.

4 Extending TM with Atomic Deferral

We support atomic deferral using two new keywords: the `deferrable` annotation on classes, and the `atomic_defer` function, which takes as arguments a function and a list of objects, each of which must be an instance of a `deferrable` class. To defer an operation, a programmer calls `atomic_defer` with a function implementing the deferred operation and a list of all the shared objects that this function may access. Fields of `deferrable` objects must not be accessed directly, but only through getters and setters (a recommended software engineering practice in any case). Thus, if o is an object with a `deferrable` class type and an *expensive* method, then we can defer the execution of that method within a transaction by writing:

$\lambda \leftarrow () \{ o.expensive() \}$
`atomic_defer`(λ, o)

The deferred operation will be executed immediately after the enclosing transaction commits, and in such a way that no other transactions can see a state that reflects the effects of the transaction but not those of its deferred operation. A deferred operation will see any effects of the transaction that occur after the call to `atomic_defer`. If `atomic_defer` is called multiple times within a single transaction, the deferred operations will be executed in the order of their respective calls to `atomic_defer`, and the effects of earlier deferred operations will be visible to later ones.

Listing 1: Implementation of atomic deferral.

```
   // Extensions to classes annotated as deferrable
   deferrable class T
     lock : TxLock // implicit per-instance lock
     ...  // programmer-defined fields
   function  transaction_safe Method(...)
       // subscribe to the implicit lock
       TxLock.Subscribe(lock)
       // programmer-defined logic
       ...

   function atomic_defer(l : λ, objs: Deferrable ...)
       // Use transaction to acquire locks without deadlock
1      transaction
2          for o : objs do
3              TxLock.Acquire(o)
4      deferred_ops.append(⟨l, objs⟩)

   Additional Per-Thread TM Metadata:
     // all deferred operations for current transaction
     deferred_ops   : list⟨λ, list⟨Deferrable⟩⟩

   function TxEnd()
       // Standard STM Commit; HTM uses a special instruction
1      ValidateReadsFinalizeWrites()
       // STM-only: ensure transaction finishes before λs run
2      Quiesce()
       // Reset thread's TM metadata
3      move(tm_free_list, local_frees)
4      move(deferred_ops, local_defers)
5      ResetLists()
       // Execute deferred operations
6      for ⟨l, objs⟩ ∈ local_defers do
7          l.execute()
8          for o ∈ objs do TxLock.Release(o)
       // Reclaim memory, reset lists
9      for ptr ∈ local_frees do free(ptr)
```

4.1 Implementing Atomic Deferral

We implement atomic deferral by using locks to protect accesses to `deferrable` objects. To provide atomicity, we acquire the locks required by the deferred operation before the transaction commits. We also need a way to notify transactions that access a `deferrable` object (directly as part of the transaction, not deferred) when the lock protecting it has been acquired (by a transaction that calls `atomic_defer` with the object): such transactions must abort and retry after the deferred operation has completed (and the corresponding locks released).

To this end, we designed *transaction-friendly locks*, which can be acquired and released within a transaction, and which provide a *subscribe* method. The subscribe method must be called from within a transaction, which blocks (or aborts) until the lock is either free or held by the subscribing thread. Multiple threads can subscribe to a lock if it is free. We describe how we implement transaction-friendly locks in Section 4.2.

Pseudocode for implementing atomic deferral appears in Listing 1. In addition to providing implementations for `atomic_defer` and `deferrable`, we modify the commit operation `TxEnd`. This code assumes that `TxLock` is a class of transaction-friendly locks, and that `Deferrable` is a base class for all `deferrable` classes.

For `deferrable` classes, we add a field that maintains a transaction-friendly lock that protects the class, and we inject a call to `TxLock.Subscribe` for this lock as the first instruction of the transaction-safe version of every member function.[1]

The `atomic_defer` function first acquires the locks of all the `deferrable` objects passed to it, and then appends all of its arguments (the function representing the deferred operation and the list of `deferrable` objects it may access) to *deferred_ops*, a thread-local list of deferred operations. This list will be used when the transaction commits, as described below.

When committing the transaction, we first proceed as usual, validating the read set, finalizing the writes, and then quiescing to avoid privatization problems. Remember that any object that might be accessed by deferred operations has already been locked (i.e., when `atomic_defer` was called with the object), so no other transaction can see any writes to it. We then execute the deferred operations in order, releasing the locks on the `deferrable` objects associated with each deferred operation after that operation is complete. (If an object is accessed by multiple deferred operations, each of them would have acquired the corresponding reentrant lock, and so it is not actually released until the last such operation completes.)

The enclosing transaction may have freed memory, which is normally deferred by the TM after the transaction has quiesced. Because deferred operations may refer to memory that was subsequently freed by the transaction, we delay the freeing of that memory a bit more, until all the deferred operations have completed.

Because deferred operations may use transactions internally, we need to make *deferred_ops* and *tm_free_list* available for their use. Thus, we copy them into local variables before executing any of the deferred operations.

To argue that this implementation is correct, that is, that a transaction and its deferred operations appear atomic, we draw an analogy with two-phase locking, a well-understood technique known to guarantee atomicity. Specifically, a transaction can be thought of as acquiring and holding a single global lock until the transaction commits. Because the lock for every object accessed by deferred operations is acquired before the transaction commits, there is an initial phase in which locks are only acquired (i.e., up to the point that the transaction commits), and a concluding phase in which locks are only released (including the implicit global lock released by the transaction on commit). So all locks are held between the time that the commit operation is invoked and the time that the commit actually occurs. We must also ensure that every access is protected by the appropriate lock, which is why the programmer must provide, when calling `atomic_defer`, all the objects that the deferred operation may access. If a deferred operation accesses some object not passed to `atomic_defer`, then a data race may occur.

4.2 Transaction-Friendly Mutex Locks

The heart of our atomic deferral mechanism is a transaction-friendly mutual exclusion lock, whose pseudocode appears in Listing 2. The `TxLock` is reentrant, storing an owner and a count of the locking depth. In this manner, a thread that holds the lock may re-acquire it by incrementing the count. Before any other thread can acquire the lock, the current owner must release the lock as often as needed to ensure $depth = 0$. Since the implementation uses transactions, the *owner* and *depth* fields need not be packed into a single machine word:

[1] STM implementations typically maintain two versions of each transaction-safe function, one that is called within transactions and one that is called when not in a transaction.

Listing 2: A transaction-friendly, reentrant mutex lock.

```
   Fields of TxLock Object:
     owner   : transaction_id   // Lock holder ID
     depth   : Integer          // For reentrancy

   function TxLock.Acquire(l)
1      atomic
           // Common case: lock is unheld
2          if l.owner = nil then
3              l.owner ← me
4              l.depth ← 1
5              return
           // Handle reentrancy/nesting
6          else if l.owner = me then
7              l.depth ← l.depth + 1
8              return
           // Else wait (spin and/or yield CPU) until lock is released
9          spin()
10         retry

   function TxLock.Subscribe()
       // Must be in transaction to call
1      if owner ≠ nil ∧ owner ≠ me then
2          retry

   function TxLock.Release(l)
1      atomic
           // [Optional] Forbid handoff of held lock
2          if l.owner ≠ me then
3              fatal error
           // Handle reentrancy/nesting
4          else if l.depth > 1 then
5              l.depth ← l.depth − 1
6              return
           // Else no reentrancy/nesting
7          l.depth ← 0
8          l.owner ← nil
```

they are only accessed within transactions. A thread that is currently in a transaction may acquire and/or release `TxLock`s, because it is correct in C++ to nest transactions. Among other things, this means that a thread can acquire multiple locks in a deadlock-free fashion, even without a global locking order: it need only issue all acquisitions inside of a transaction.

`TxLock`s are elidable within transactions, via the `Subscribe` method: a transaction that subscribes to a `TxLock` blocks until the lock is either unheld, or held by the calling thread. Subscription only reads the *owner* field, which allows concurrent subscription by multiple threads. When any thread acquires the `TxLock`, all subscribing transactions will conflict with the new lock owner, and will abort. When the `TxLock` is acquired, the C++ TMTS ensures correct fence semantics: since the transaction accesses shared memory, the TM implementation is required to guarantee that memory accesses preceding the transaction order before it, and memory accesses following the transaction order after it.

If the C++ TMTS adds efficient support for `retry`, transactions could yield the CPU if they attempt to acquire or subscribe to a lock that is held by another thread, and would be woken automatically when the lock is released. In the meantime, we implement `retry` by placing an `atomic` transaction inside a while loop, replacing the `retry` instruction with an exception throw, and adding a `break` as the last statement in the transaction.

Listing 3: Diagnostic logging from a critical section.

```
// Version with irrevocable transactions
synchronized
    // x is a mutable string
    // i is a mutable integer
    ...
    fprintf(stderr, str, x, i)
    free(x) // Optional
    ...
```

```
// Deferrable for the log file's descriptor
class defer_fprintf: public Deferrable
    fd : file    // output file descriptor
    ...

// global instance of log file descriptor
df : defer_fprintf(fd ← stderr)
```

```
// Version with atomic_defer
atomic
    ...
    tmp ← sprintf(str, x, i)
    λ ←()
        fprintf(df.fd, tmp)
        free(tmp)
        free(x) // Optional
    atomic_defer(λ, df)
    ...
```

4.3 Practical Concerns

Deferring operations creates a nonlinear control flow within a program. This nonlinearity is not observable to concurrent threads: the transaction and its deferred operations appear to be a single, serializable operation. However, within the transaction, the programmer must be mindful of a few challenges. First, the state of the object and thread-private data at the time when the `atomic_defer` keyword appears is not immutable, and may change in the suffix of the transaction that executes before the deferred operation. In addition, the deferred operation does not execute transactionally, and thus races can occur if the deferred operation accesses shared data not protected by the associated `TxLock`s. Second, the programmer must encapsulate shared objects carefully. Consider a deferred operation that performs a `write` of byte stream B to file descriptor F. If F is shared, then it should be a field of a `Deferrable` object. If B is shared, then it, too, should be a field of a `Deferrable` object. Programmers must decide if B and F should be fields of the same `deferrable` object, or of multiple objects. Third, since system calls made within a deferred operation happen immediately, some possibility for performance bottlenecks remains. For example, an `fsync` within a deferred operation is often necessary. With `atomic_defer`, the `fsync` and any associated error recovery can be atomic with the transaction, and will not block *all* transactions. However, lengthy deferred operations will still block concurrent transactions that call a method of the associated `deferrable` objects.

5 Programming With Atomic Defer

We now present examples of `atomic_defer` in real applications. The examples depict common use cases, and show the deferral of increasingly complex operations without sacrificing atomicity or resorting to serialization.

5.1 Basic Logging

In programs such as memcached [18] and Atomic Quake [27], critical sections occasionally perform logging operations, such as error messages and diagnostic writes to per-thread logs. The program does not require any ordering among logging operations: they are timestamped, and the order can be determined post-mortem. The return values of the output operations are typically ignored. An example appears in Listing 3.

When the values to be logged (`x` and `i`) are mutable shared data, existing programs resort to irrevocability or they skip the logging operation. When the values can be encapsulated in a `Deferrable` object, `atomic_defer` is a straightforward transformation: the output string is prepared within the transaction, and the output is deferred until the end of the transaction.

Listing 4: Durable output with guaranteed order.

```
// Deferrable wrapper for file descriptors
class defer_fd: public Deferrable
    fd : file     // output file descriptor
    ...

// Deferrable objects
fd_D1 : defer_fd(fd ← ...)
buff_D1 : defer_buffer(buf ← ..., flag ← false)
// Durable output to fd_D1
atomic
1   ...
2   λ ←()
3       write(fd_D1.fd, buff_D1.buffer)
4       fsync(fd_D1.fd)
5       buff_D1.flag ← true
6   atomic_defer(λ, fd_D1, buf_D1)

// Deferrable wrapper for output buffer
class defer_buffer: public Deferrable
    buf : buffer     // buffer data
    flag: boolean    // is buffer written?

// Deferrable objects
fd_D2 : defer_fd(fd ← ...)
buff_D2 : defer_buffer(buf ← ..., flag ← false)
// Conditional durable output to fd_D2
atomic
7   Subscribe(buff_D1)
8   if buff_D1.flag then
9       λ ←()
10          write(fd_D2.fd, buff_D2.buffer)
11          fsync(fd_D2.fd)
12          buff_D2.flag ← true
13  atomic_defer(λ, fd_D2, buf_D2)
```

Note that this approach ensures ordering of all logging operations on the encapsulated file descriptor. A simpler approach, when ordering is not needed, is to pass **nil** as the second argument on line 8. This approach causes serialization only among transactions that use *df*.

Because transactional versions of existing programs tend to omit this instrumentation in order to avoid serialization, we did not observe a performance impact when applying `atomic_defer` to memcached. However, atomic deferral keeps the code robust and complete without adding too much burden on programmer, and it makes it easier to debug programs during development, by enabling non-serializing `printf` debugging.

5.2 Durable Output

Programs often rely on the `fsync` system call to persistent output. In some cases (e.g., durable database operations), it is necessary to order outputs based on the timing of `fsync` calls, such that when there are two files: F_1 and F_2, $F_2$is not updated until after F_1's updates have reached the disk. Simply deferring an `fsync` operation in this case is insufficient. With `atomic_defer`, we can encapsulate the completion status of the `fsync` in a `Deferred` object that is associated with the deferred `fsync` operation.

In Listing 4, one thread executes the transaction (T_1) on lines 1 to 6, and another executes the transaction (T_2) on lines 7 to 13. We wish to ensure that T_2 does not write $buff_{D2}$ to file fd_{D2} unless T_1's write of $buff_{D1}$ to file fd_{D1} has been persisted to disk. Since the flag indicating the completion of T_1's `fsync` is encapsulated in a `Deferrable` object, and T_1 sets that flag in an operation that has been deferred, we know that $buff_{D1}$'s implicit lock will be held during the time that the flag is set, and will not be released until after the `fsync` returns. Consequently, when T_2 executes line 7, three cases are possible: (1) T_1 has not yet executed line 6, in which case line 7 returns, and then line 8 evaluates to false; (2) T_1 has executed line 6 but has not completed lines 3 to 5, in which case T_2 will call `retry` and ultimately land in the third case; or (3) T_1 has completed its deferred execution of lines 3 to 5, in which case T_2 can subscribe to $buff_{D1}$, and then observe a **true** value on line 8. Note that T_2 can only perform its write in the third case, which orders lines 3–5 before lines 10–12, and thus the deferred outputs occur and reach the disk in the required order.

Listing 5: MySQL critical sections in file pool management that are used in asynchronous I/O.

```
   // atomic_defer: types and variables
   class file_system_t: public Deferrable
     ...
     space_list : file_space_t

   // wrap the file system as a deferrable object
   file_system : file_system_t

   mySQL_initialize (...)
     // open tablespace data files
1    atomic
       ...
2      λ ←()
3        for space ∈ space_list
4          for node ∈ space
5            node ← open(...)
6      atomic_defer(λ, file_system)
         ...
   mySQL_destroy (...)
     // close tablespace data files
7    atomic
       ...
8      λ ←()
9        for space ∈ space_list
10         for node ∈ space
11           close(node)
12     atomic_defer(λ, file_system)
         ...

   mySQL_io_prepare (...)
13   close_more :
14   atomic
       // check system states and select files
       ...
15     λ ←()
16       if close(file) = −1
17         error
18       n_open ← n_open − 1
19       if (n_open ≥ max_n_open)
20         need_close ← true
21         goto end
         // check the node to do I/O
22       if ¬node.open
           // get file size, do an open and close
           // save metadata for future I/O
23         if node.size = 0
24           node ← open()
25           offset ← lseek(file, 0, SEEK_END)
26           success ← pread(two pages)
27           close(node)
28         node ← open()
29       end :
30     atomic_defer(λ, file_system)
         ...
31   if (need_close)
32     goto close_more
```

5.3 Opening Files as Output

Our final example comes from the MySQL InnoDB storage engine. InnoDB maintains a pool of file descriptors, which is protected by a lock. Metadata is associated with each file

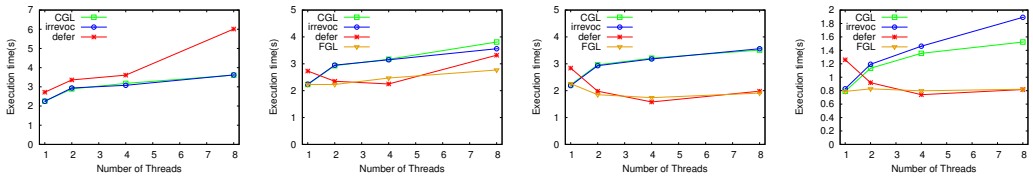


Figure 2 Atomic_defer performance on an I/O microbenchmark (a) - (d).

descriptor, and allows updates to files to be performed via asynchronous I/O. For example, to append new records to the end of a file, a thread locks the pool, updates the size of the file, and then unlocks the pool. It then issues an asynchronous write. Subsequent appends will follow the same protocol, and hence will appear at a later point in the file, even if their writes reach the disk earlier.

While reads and writes do not occur in critical sections, and hence would not serialize a transactional version of InnoDB, the management of the pool depends on the ability to open and close files dynamically, in order to stay below a pre-set maximum number of open files. If a file must be opened when the pool is at capacity, then a thread will lock the pool, close some other files that do not have outstanding accesses in-flight, and then open the new file. In transactional InnoDB[2], this operation requires irrevocability, and serializes all memory transactions, to include those performing read-only queries of data within the database.

With `atomic_defer`, the pool becomes a `Deferrable` object. On any modification to file descriptor metadata, a thread uses a transaction that subscribes to the pool. Thus, file operations can proceed fully in parallel, since they use asynchronous I/O to perform their file accesses, and transactions to operate on disjoint file metadata regions. In the uncommon cases where files are opened and closed, the system calls are deferred from a transaction. While the system calls are in-flight, concurrent accesses to the pool stall (via `retry`). Once the pool is returned to a usable state, any suspended threads resume.

6 Performance Evaluation

We now present experiments that demonstrate the benefit of `atomic_defer`. We conduct tests on two platforms. In charts depicting scalability up to 8 threads, the platform is a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM, includes 8 GB of RAM, and runs a Linux 4.3 kernel. Experiments with larger thread counts were conducted on a machine with two 18-core/36-thread Intel E5-2699 V3 CPUs running at 2.30GHz. This CPU also supports TSX, includes 128 GB of RAM, and runs a Linux 4.8 kernel. Our extensions were implemented in GCC 5.3.1. Results are the average of 5 trials.

6.1 Performance of atomic_defer on a Transactional I/O Microbenchmark

One motivation for `atomic_defer` is to avoid the serialization of `synchronized` transactions, while allowing output that is atomic with respect to the transaction. We begin with a microbenchmark study to observe the behavior of transactions that perform irrevocable operations on files. Our microbenchmarks are patterned after work by Demsky and Tehrany [3].

[2] Unfortunately, adding TM to InnoDB revealed a bug in GCC, which produces an internal compiler error.

Listing 6: An example of deferring I/O and system calls.

```
   // Encapsulate streams in a Deferrable object
   class defer_file: public Deferrable
       input    // input stream
       output   // output stream

   // An array of files
   dfs: defer_file[]

   // Operation to be deferred
1  λ ←(id, content)
       // Read File
2      if ¬dfs[id].input.open() then error
       // Get the length of the file
3      dfs[id].input.seekg(0, end)
4      len ← dfs[id].input.tellg()
5      dfs[id].input.close()
       // Write to the file and close
6      tmp ← format(content, len)
7      dfs[id].output.write(tmp)
8      dfs[id].output.close()

   // Irrevocable version of benchmark
1  synchronized
2      content ← ...
3      id ← ...
4      λ(id, content)

   // atomic_defer version of benchmark
1  atomic
2      content ← ...
3      id ← ...
4      atomic_defer(λ(id, content), dfs[id])
```

Whereas they required custom instrumentation of system calls in order to make them transaction safe, we run I/O operations without instrumentation, using either irrevocability or `atomic_defer`. Listing 6 presents the general behavior of our microbenchmarks: a transaction produces content and identifies a file to update. It then performs I/O, which includes opening a file, reading the file length, and appending data to the file. The I/O can be deferred or executed irrevocably. To use `atomic_defer`, we encapsulate the I/O streams in `deferrable` objects, and then use `atomic_defer` to delay the operation on line 5. Figure 2 presents experiments with four configurations of the microbenchmark. In each case, threads cooperate to complete a total of 1M operations. The figure presents results for STM, but trends for HTM are the same.

Figure 2a explores the overhead of atomic deferral when there is only one file, and hence no concurrency, by comparing performance when transactions are replaced with a coarse-grained lock (CGL), and when transactions use irrevocability (irrevoc), or atomic deferral (defer). We see that the baseline GCC TM implementation (irrevoc) is well-tuned to handle irrevocability: it serializes transactions early, avoids instrumentation, and achieves performance comparable to CGL. In contrast, `atomic_defer` pays a constant overhead per transaction to support rollback, even though no rollbacks occur. As the thread count increases, overheads due to `retry` cause additional slowdown. This is partly a result of our `retry` implementation aborting and immediately retrying, instead of de-scheduling the transaction until it can make progress. Until the C++ TMTS includes efficient `retry`, this cost is unavoidable.

Figures 2b and 2c expand the number of files to 2 and then 4, and threads update files with equal probability. We include another non-transactional baseline, with one fine-grained lock (FGL) per file. We again see that single-threaded code has higher overhead when using `atomic_defer`, due to instrumentation and the management of lambdas. While the behavior

Listing 7: Deferring reliable output in PARSEC dedup.

```
   function pipeline_out(buf, len, fd)
       // fd may be unreliable, so monitor progress of
         writes
1      (p, nsent, rv) ← (buf, 0, 0)
2      while nsent < len
3          rv ← write(fd, p, len − nsent)
4          if transient_error(rv) then
5              continue
6          if fatal_error(rv) then error
7          nsent ← nsent + rv
8          p ← p + rv
9      fsync(fd)
10     free(buf)
```

```
   // Version with irrevocable transactions
1  synchronized
       ...
2      pipeline_out(packet.buf, len)
       ...

   // Version with atomic_defer: packet is now
     deferrable
   deferrable class packet
1  atomic
       ...
2      λ ←()
3          pipeline_out(packet.buf, len)
           ...
4      atomic_defer(λ, packet)
```

of CGL and irrevoc is unchanged, deferral now shows scaling on par with fine-grained locks, achieving indistinguishable performance at 2 and 4 threads. When the thread count greatly exceeds the potential concurrency (e.g., 8 threads and 2 files, Figure 2b), we still see extra overheads from `retry`. However, when there is enough concurrency in the workload (e.g., 4 files), `atomic_defer` scales well.

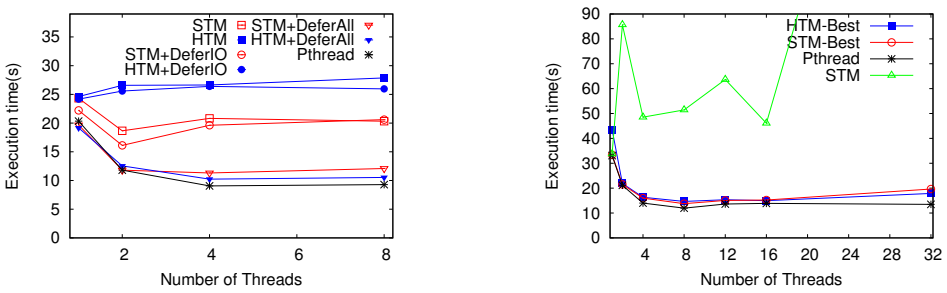
Finally, in Figure 2d transactions append to files that are kept open throughout the experiment. There are still 4 files, but the smaller critical sections reveal an overhead in the irrevocability mechanism: when one transaction becomes irrevocable, the others block, possibly yielding the CPU. When the irrevocable transaction is brief, the overhead of yielding becomes visible, and irrevoc degrades worse than CGL. Meanwhile, FGL has flat performance, and defer overcomes latency at 1 thread to be competitive with FGL.

6.2 Performance of atomic_defer on PARSEC Dedup

Wang et al. reported [21] that PARSEC's dedup kernel [1] ceased to scale when transactions replaced locks. Dedup is a pipeline application, and the original file output stage performs output while holding a lock; Wang's version replaces that lock with an irrevocable transaction. When the irrevocable transaction executes, it must serialize all concurrent transactions.

When we rewrote dedup's output operation to use `atomic_defer`, irrevocability ceased to cause performance degradation, but the benchmark still scaled poorly. A sketch of the code transformation appears in Listing 7. Since the buffer to be output was already encapsulated in a struct ("packet"), we made that struct `deferrable` and ensured that its fields were accessed through getters and setters. Deferring the operation was then a one-line change, which preserved the ordering of `fsync` operations and error handling with respect to output and subsequent concurrent accesses. The performance of dedup with this change appears in Figure 3 (a), as "+DeferIO".

We discovered that the `Compress` function was marked as `pure`, because it does not access any shared memory. Marking the function `pure` indicates to the compiler that the function can be run without instrumentation, lacks side effects, and can be run from a non-irrevocable context even when the compiler cannot prove that irrevocability is not needed. `Compress` is a long-running function, and in HTM, it accesses more memory than can be tracked by the HTM; the HTM execution serializes whenever a call to `Compress` exceeds the capacity of the hardware. In STM, the call to `Compress` represents a period of time during which other transactions can commit, but will delay in their quiesce operations. While the run-time behaviors are different, the consequence is the same: when one transaction calls `Compress`, other transactions cannot make progress.



(a) Effects of `atomic_defer` on I/O and pure functions

(b) Overall performance

Figure 3 Performance of PARSEC dedup with `atomic_defer`.

Since `Compress` is pure, it can be deferred. We encapsulated the compressed buffer as a `deferrable` object, so that the run-time system can suspend transactions when they attempt to access a buffer that is locked for deferred compression. This has a profound impact on both STM and HTM. In HTM, the transaction ceases to overflow hardware capacity, and serialization is avoided. In STM, compression ceases to impede quiescence, and concurrent threads can make forward progress. In Figure 3(a), we see that the "+DeferAll" curves for both HTM and STM now compete with pthread locks, representing a 1.7x speedup for STM and 2.7x speedup for HTM.

Lastly, we measure the impact of `atomic_defer` on the 36-core system, to see how performance scales across chips and in the face of significantly more hardware parallelism. In Figure 3 (b), the performance of the baseline HTM is not shown: the 32-thread STM performance exceeds 270 seconds, and HTM never scales. When we employ `atomic_defer` to move output and pure functions out of transactions, both STM and HTM improve by roughly 10x compared to their respective TM baselines, reaching the same performance as pthread locks. While these optimizations require more careful reasoning about the program, we contend that these optimizations are still easier than reasoning about fine-grained locking.

7 Related work

Atomic deferral is a general mechanism for moving code out of transactions, but retaining serializability, through the composition of TM with two-phase locking. As we have seen in our examples and evaluation, atomic deferral offers an implementation-agnostic technique for performing output operations and other system calls within transactions. It also enables the movement of costly operations outside of the constrained environment presented by a general-purpose TM. Our work bears resemblance to, and is inspired by, several prior works in the areas of deferral, transactional system calls, irrevocability, and escape actions. Below, we summarize the most significant relationships.

Support for Deferred Operations. Language-level support for deferred system calls was first proposed by Carlstrom et al. [2], and expanded by Ni et al. [16]. These proposals took a broad approach to deferral, and considered deferring operations for both committed and aborted transactions. However, these deferred operations do not run atomically with the parent transaction, nor can they access transactional data without either copying or relying on ad-hoc synchronization in the deferred operation. Relative to these works, our contributions are the addition of locks and the introduction of a two-phase locking pattern that ensures serializability.

Rossbach et al. [17] were first to use locks to coordinate transactions and non-transactional code. Volos et al. [20] followed with a comprehensive approach to deferral focused on enabling transactional system calls (including I/O). Volos extended the OS with "sentinel" locks, which allowed software transactions to exclusively access file descriptors and other resources. Using these sentinel locks, output operations could appear to execute in the context of a transaction, but actually run after the transaction committed. On the one hand, Volos's work is more comprehensive than ours, as it deals with a wide array of system calls, and requires less programmer effort to perform transactional I/O. On the other hand, our work is more practical: it does not require a deadlock detection algorithm within the OS, or any OS modifications; it is free of system calls, and hence compatible with HTM (today's HTM systems abort on any change of privilege); and it allows the programmer to control the granularity at which operations are serialized (e.g., in the case of MySQL's file descriptor pool, where one lock abstractly covers an unbounded set of file descriptors). Our approach also makes it easier to handle timing and errors in deferred operations; we are not aware of a way to use Volos's work to achieve the behaviors in Listings 4 and 5.

Irrevocable Operations. Prior work has encouraged the broad use of irrevocability [16] for transactional I/O, contention management, and even as a performance optimization for to reduce logging overheads in long-running STM transactions. Clearly atomic deferral does not obviate irrevocability. In particular, our work assumed that the continuation of a transaction does not depend on the result of the deferred operation. For output to unreliable media (e.g., Listing 7), atomic deferral is sufficient. However, if the error on line 6 ought to influence operations after line 4 of the transaction on the right side of the listing, then irrevocability is the only known solution.

Escape Actions. Another approach to reducing the costs targeted by atomic deferral is the use of escape actions. These may be ad-hoc [26], or formalized as open nesting [15] or transactional boosting [8]. Like atomic deferral, these mechanisms provide a way to avoid logging overhead in complex operations, and also to perform I/O operations within transactions. However, these techniques are rarely compatible with HTM [13] (an exception is the IBM POWER TM [14]). Additionally, in STM, these techniques reduce the transactional footprint, but still run in the context of an active transaction; consequently, they retain the quiescence-associated delays shown in Figure 1. These techniques also require ad-hoc compensating actions and error handlers. On the other hand, boosting techniques offer great promise in areas where atomic defer is not useful, such as the creation of highly concurrent data structures [7, 22].

8 Conclusions and Future Work

In this paper, we presented a technique for atomically deferring operations in memory transactions. The key feature of our work is that concurrent transactions cannot detect that an operation was deferred: the operation appears atomic with the corresponding transaction, which retains serializability. The fundamental technique to enable atomic deferral is composing transactions with locks and `retry`-based condition synchronization, to facilitate a form of two-phase locking. With the deferred operation, transactions may perform complex operations and access a subset of shared memory. Using atomic deferral allows transactions to perform output without serializing, and was the foundation for a dramatic improvement in the performance of the PARSEC dedup benchmark.

Atomic deferral requires more complex reasoning by programmers than irrevocability, and is less general. However, when applicable, it eliminates serialization overheads, and shortens the time that transactions spend quiescing. In our view, the additional programmer overhead to use atomic deferral is small, and more than justified by the benefits. For example, we presented a scenario where atomic deferral can avoid serialization when managing file descriptor pools in MySQL, and another where files can be updated in order, while obeying strong persistence requirements.

As future work, we are interested in tools for automatically transforming output operations into deferred operations, and studying the relationship between atomic deferral and nested transactions. We are also interested in crafting a more formal correctness argument, which may influence the use of transaction-friendly locks in a greater range of workloads.

Acknowledgements. We thank Dave Dice and Tim Harris for their advice and guidance during the conduct of this research. At Lehigh University, this work was supported in part by the National Science Foundation under Grant CAREER-1253362, and through financial support from Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1 Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th PACT*, Toronto, ON, Canada, oct 2008.

2 Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th PLDI*, jun 2006.

3 Brian Demsky and Navid Tehrany. Integrating File Operations into Transactional Memory. *Journal of Parallel and Distributed Computing*, 71(10):1293–1304, 2011.

4 Nuno Diegues, Paolo Romano, and Luis Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd PACT*, Edmonton, AB, Canada, aug 2014.

5 Free Software Foundation. Transactional Memory in GCC [online]. 2016. URL: `http://gcc.gnu.org/wiki/TransactionalMemory`.

6 Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th PPoPP*, Chicago, IL, jun 2005.

7 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Optimistic Transactional Boosting. In *Proceedings of the 19th PPoPP*, Orlando, FL, feb 2014.

8 Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th PPoPP*, Salt Lake City, UT, feb 2008.

9 Maurice P. Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*, San Diego, CA, 1993.

10 ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory, May 2015.

11 Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th HPCA*, Orlando, FL, February 2014.

12 Matthew Kilgore, Stephen Louie, Chao Wang, Tingzhe Zhou, Wenjia Ruan, Yujie Liu, , and Michael Spear. Transactional Tools for the Third Decade. In *Proceedings of the 10th TRANSACT*, Portland, OR, jun 2015.

13 Yujie Liu, Stephan Diestelhorst, and Michael Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th SPAA*, Pittsburgh, PA, jun 2012.

14 Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd ISCA*, Portland, OR, June 2015.

15 Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Rick Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th PPoPP*, San Jose, CA, mar 2007.

16 Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd OOPSLA*, Nashville, TN, USA, 2008.

17 Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and Managing Transactional Memory in an Operating System. In *Proceedings of the 21st SOSP*, Stevenson, WA, oct 2007.

18 Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th ASPLOS*, Salt Lake City, UT, mar 2014.

19 Michael Spear, Virendra Marathe, Luke Dalessandro, and Michael Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th PODC*, Portland, OR, 2007.

20 Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael Swift, and Adam Welc. xCalls: Safe I/O in Memory Transactions. In *Proceedings of the EuroSys2009 Conference*, Nuremberg, Germany, March 2009.

21 Chao Wang, Yujie Liu, and Michael Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th SPAA*, Prague, Czech Republic, jun 2014.

22 Lingxiang Xiang and Michael Scott. Software Partitioning of Hardware Transactions. In *Proceedings of the 20th PPoPP*, San Francisco, CA, feb 2015.

23 Tingzhe Zhou, Victor Luchangco, and Michael Spear. Brief Announcement: Extending Transactional Memory with Atomic Deferral. In *Proceedings of the 29th SPAA*, Washington DC, July 2017.

24 Tingzhe Zhou and Michael Spear. The Mimir Approach to Transactional Output. In *Proceedings of the 11th TRANSACT*, Barcelona, Spain, mar 2016.

25 Tingzhe Zhou, PanteA Zardoshti, and Michael Spear. Practical Experience with Transactional Lock Elision. In *Proceedings of the 46th ICPP*, Bristol, UK, August 2017.

26 Craig Zilles and Lee Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. In *Proceedings of the 1st TRANSACT*, Ottawa, ON, Canada, jun 2006.

27 Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th PPoPP*, Raleigh, NC, feb 2009.

Lock Oscillation: Boosting the Performance of Concurrent Data Structures[†]

Panagiota Fatourou[1] and Nikolaos D. Kallimanis[2]

1 **Institute of Computer Science (ICS), Foundation of Research and Technology-Hellas (FORTH), and Department of Computer Science, University of Crete, Greece**
faturu@csd.uoc.gr

2 **Institute of Computer Science, Foundation of Research and Technology-Hellas, Crete, Greece**
nkallima@ics.forth.gr

Abstract

In *combining-based synchronization*, two main parameters that affect performance are the *combining degree* of the synchronization algorithm, i.e. the average number of requests that each *combiner* serves, and the number of expensive synchronization *primitives* (like `CAS`, `Swap`, etc.) that it performs. The value of the first parameter must be high, whereas the second must be kept low.

In this paper, we present Osci, a new combining technique that shows remarkable performance when paired with cheap context switching. We experimentally show that Osci significantly outperforms all previous combining algorithms. Specifically, the throughput of Osci is higher than that of previously presented combining techniques by more than *an order of magnitude*. Notably, Osci's throughput is much closer to the ideal than all previous algorithms, while keeping the average latency in serving each request low. We evaluated the performance of Osci in two different multiprocessor architectures, namely AMD and Intel.

Based on Osci, we implement and experimentally evaluate implementations of concurrent queues and stacks. These implementations outperform by far all current state-of-the-art concurrent queue and stack implementations. Although the current version of Osci has been evaluated in an environment supporting user-level threads, it would run correctly on any threading library, preemptive or not (including kernel threads).

1998 ACM Subject Classification D.1.3 Programming Techniques - Concurrent Programming

Keywords and phrases Synchronization, concurrent data structures, combining

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.8

1 Introduction

The development of efficient parallel software has become a necessity due to the dominance of multicore machines. One obstacle in achieving good performance when introducing parallelism in modern applications comes from the synchronization cost incurred by those parts of the application that cannot be parallelized. Efficient synchronization mechanisms are then required to maintain this synchronization cost low.

[†] This work has been supported by the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the ExaNoDe project (grant agreement 671578) and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project.

Whenever a parallel application wants to access shared data, a synchronization request is initiated. In order to avoid races, these requests must be executed in mutual exclusion. Therefore, a lower bound on the time to execute m such requests is the time it takes for a single thread to sequentially execute them, sidestepping the cost of the synchronization protocol. An *ideal* synchronization protocol would not require more time than this, independently of the number of the active threads and despite any contention on the accessed data. In practice, even the best current synchronization protocols cause a drastic reduction in performance, even in low contention.

Recent work [8, 9, 14] has focused on developing synchronization protocols implementing the *combining* synchronization technique (first realized in [11, 31]). This technique has been argued [8, 9, 14] to be very efficient. A combining synchronization protocol maintains a list to store the pending synchronization requests issued by the active threads. A thread first announces its request by placing a node in the list, and then tries to acquire a global lock. If it does so, it becomes a *combiner*, and serves not only its own synchronization request, but also active requests announced by other threads. Each thread that has not acquired the lock, busy waits until either its request is executed by a combiner or the global lock is released.

CC-Synch [9] is a simple implementation of the combining technique which outperforms previous state-of-the-art combining algorithms including flat-combining [14] and OyamaAlg [26], and other synchronization mechanisms such as CLH-locks [7, 19] and a simple lock-free algorithm [9]. Nevertheless, Figure 1 indicates that the performance of CC-Synch is still far from the ideal in a multicore machine equipped with 64 processing cores (more details on this experiment are provided in Section 5); the ideal performance is measured by calculating the time that it takes to a single thread to execute the total number of synchronization requests (that are to be executed by all threads) sidestepping the synchronization protocol, as well as the local work that follows each of the synchronization requests that it has to perform itself.

In this paper, we present a technique that significantly enhances the performance of combining-based synchronization by reducing the number of expensive synchronization primitives such as `Compare&Swap` (`CAS`) or `Swap` that are performed on the same shared memory location, resulting in much fewer cache-misses and cpu backend stalls. Yet, this technique results in algorithms that are as simple as using CC-Synch or any other synchronization technique. The technique enables batching of the synchronization requests initiated by threads running on the same core. The requests are batched in a single "fat" node, which is then appended in the list of the announced synchronization requests by performing just a single expensive synchronization primitive.

We study the impact on performance of this technique when combined with cheap context switching. Specifically, we experimentally argue that its performance power is remarkable when employed in an environment supporting user-level threads. We present Osci, a new combining synchronization protocol which exploits this technique. Osci exhibits performance that is surprisingly closer to the ideal than previous combining algorithms. We experiment with Osci in a setting where a kernel-level thread running on each core has spawned a number of user-level threads. Osci appropriately schedules the threads, using a fair implementation of `Yield` (whenever a `Yield` is executed, the running thread voluntarily gives the control of the core that it is running to some other thread), to achieve better performance.

Osci ensures that a thread p from the set P_c of threads running on a core c, initializes the contents of a node nd and informs other threads in P_c that they can record their requests there. So, p initiates a recording period for nd. Next time p is scheduled, it informs threads in P_c that the recording period for nd is over and appends nd to the shared list of requests. Thus, when nd is appended in the list, more than one requests by threads in P_c may have been recorded in it. The combiner traverses the request list and serves all requests recorded

Table 1 Summary of the known combining techniques.

Algorithm	Primitives	Scheduling-Aware
Blocking Algorithms		
OyamaAlg [26]	`CAS`, `read-write`	No
flat-combining [14]	`CAS`, `read-write`	No
CC-Synch [9]	`Swap`, `read-write`	No
DSM-Synch [9]	`Swap`, `CAS`, `read-write`	No
Osci (this paper)	`Swap`, `CAS`, `read-write`	Yes
Wait-Free Algorithms		
PSim [8]	`LL/SC`, `read-write`, `Fetch&Add`	No
PSim-x (this paper)	`LL/SC`, `read-write`, `Fetch&Add`	Yes

in each of the list nodes. Each of the other threads performs local spinning until either its request is served by a combiner or the thread becomes the new combiner.

In modern NUMA architectures, the execution of a synchronization primitive (`CAS`, `Swap`, etc.) on a shared memory location causes expensive cache misses and costs at least a few thousands of cpu cycles. Specifically, such primitives usually involve a flooding of invalidation messages performed by the coherence protocol [17] to those cores that keep a copy of the contents of the shared memory location in their caches (`read` is usually cheaper since it avoids causing any invalidation messages). In contrast to the expensive synchronization primitives, a context switch (`Yield`) among user-level threads running on the same core may cost no more than a hundred cpu cycles. In most locking and combining protocols, every request issued by a thread performs at least one expensive synchronization instruction on a common shared variable, resulting in a cache-line invalidation. For instance, in a queue lock, this shared variable is the Tail of the queue. Thus, the application (or the announcement) of k requests results in (at least) k cache invalidations, causing contention and increased traffic on the interconnection network. By using cheap context switching, Osci creates fat nodes containing batches of requests. It does so by attempting to pass the control of the processor to all threads that are running on the same core, thus enabling them to announce new requests at a very low cost. Each fat node may contain up to $t > 1$ requests (all issued by the user-level threads that are running on the same core). In this way, Osci substantially reduces the synchronization cost for announcing and applying batches of requests. Specifically, k requests may cause only k/t cache line invalidations for their announcement, resulting to substantially reduced coherence traffic on the interconnection network. As an immediate consequence, Osci does not only achieve high combining degree but it also causes the smallest amount of cache misses (all cache levels), due to low number of cache invalidations, and the smallest amount of backend stalls than all the other protocols (see Section 5 for a more detailed performance analysis).

We experimentally compare Osci with known synchronization algorithms, i.e. CC-Synch [9], PSim [8], flat-combining [14], OyamaAlg [26], a blocking `Fetch&Multiply` implementation based on CLH spin-locks [7, 19], and a simple lock-free implementation (see Table 1). (MCS locks [21] exhibit similar performance to CLH locks.) Osci outperforms CC-Synch by a factor of up to 11x (Figure 4b) without increasing the latency of CC-Synch. The performance advantages of Osci over all other algorithms are even higher. It is noticeable that Osci's performance is not worse than that of CC-Synch when context switching is expensive (e.g. in systems that do not support user-level threads). It is worth noting that employing user-level threads in previous algorithms does not significantly improve their performance (see Table 2).

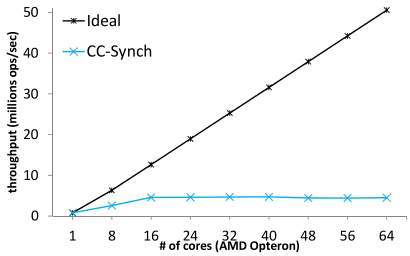


Figure 1 Average throughput of the ideal implementation, and the CC-Synch [9] implementation of a `Fetch&Multiply` object on a 64-core machine.

Osci is linearizable [15] (see Section 4 for a sketch of proof). *Linearizability* ensures that in every execution α, each synchronization request executed in α, appears to take effect, instantaneously, at some point in its execution interval.

We used Osci to get an implementation of a queue (OsciQueue) and a stack (OsciStack). Experiments show that OsciQueue outperforms all current state-of-the-art implementations of queues (combining-based or not), including LCRQ [25], CC-Queue [9], SimQueue [8], and the lock-free queue in [22]. Section 5 reveals that OsciQueue is more than 4 times faster than LCRQ [25], the state-of-the-art concurrent queue implementation. OsciStack exhibits performance advantages similar to that of OsciQueue.

We also present PSim-x, a simple variant of PSim [8]. PSim is a practical universal construction which implements the combining technique in a *wait-free* manner (*wait-freedom* ensures that each active thread manages to complete the execution of each of its requests in a finite number of *steps*). PSim can simulate any concurrent object given that only a small part of the object's state is updated by each request. In environments providing fast context switching, PSim-x achieves a significantly increased combining degree in comparison to PSim. This is done by using oversubscribing and applying appropriate scheduling on threads. As a result, the performance of PSim-x is highly enhanced compared to that of PSim. Specifically, PSim-x outperforms previous synchronization techniques (other than Osci). It also improves upon Osci by being wait-free (assuming that failures occur at the core level rather than at the thread level). Its performance, albeit lower than that of Osci, is still close to the ideal. The same holds for PSimQueue-x, a concurrent queue implementation that is based on PSim-x.

It is noticeable that based on PSim-x, it is straightforward to implement, in a wait-free manner and at a very low cost, useful complex synchronization *primitives* such as `CAS` on multiple words (and many others), that are not provided by current architectures.

2 Related Work

The current state-of-the-art combining algorithm is CC-Synch [9]. CC-Synch employs a single list to (1) store the synchronization requests and (2) implements the lock as a fast queue-based CLH-like spin-lock [7, 19]. This made CC-Synch simpler than previous protocols [14, 26]. Moreover, CC-Synch causes a bounded number of cache misses per request and its *combining degree* (i.e. the average number of requests served by a combiner) is argued [9] to be higher than that of previous techniques. Osci shares some ideas with CC-Synch. However, Osci applies a different technique from that of CC-Synch for announcing requests, batching them in fat nodes before placing them in the list of the pending requests, and it features an additional synchronization layer for the coordination of user-level threads in a way that the algorithm works even with preemptive schedulers.

A hierarchical version of CC-Synch, called H-Synch [9], exploits the hierarchical communication nature of some systems which organize their cores into clusters and provide fast

communication to the threads running on the same cluster, and much slower communication among threads running on cores of different clusters. Our experiments that have been conducted on (a) an AMD machine equipped with 4 AMD Opteron 6272 processors, and on (b) an Intel machine equipped with 4 Intel Xeon E5-4610 processors, show that H-Synch does not perform much better than CC-Synch in this kind of machines (similar results are also presented in [9]). However, batching the requests of the threads running on the same core can be considered as a form of performing hierarchical combining with the cores playing the role of clusters. We have experimented with a variant of H-Synch which employs many user-level threads per core and its performance is much worse than Osci (almost 2.5x slower). The reason is that Osci applies combining in two levels, among the threads of a core and across cores, whereas H-Synch uses a lock to synchronize threads across cores.

Although the employed thread model has been studied in other contexts [5, 30], to the best of our knowledge, this is the first paper that studies the performance impact of fast context switching in the context of combining-based synchronization. Fast context switching is realized here by employing user-level threads. However, our algorithms can be utilized efficiently in several other contexts, like, the Glasgow Haskell compiler [20], applications based on several JAVA implementations [1, 23], OpenMP tasks [12], operating systems that support scheduler activations [3], applications using User-Mode scheduling provided by the recent versions of the Windows operating system [2], and tasks in Cilk-like environments [10]. The use of `Yield` does not play any role in the correctness of Osci and PSim-x. So, they could work efficiently in any environment that exhibits fast context switching, preemptive or not (even if the thread scheduling decisions were made by the operating system [3]).

Blocking implementations of the combining technique are presented in [9, 14, 26]. All are outperformed by CC-Synch [9] and PSim [8]. Sagonas *et. al* [18] have designed a combining technique, optimized for concurrent objects that support operations, which do not require to return some value (e.g. the ENQUEUE operation of a concurrent queue). Their experiments show that their implementation outperforms CC-Synch and flat-combining in that case.

Holt *et. al* [16] present a generic framework based on flat-combining [14] suitable for systems that communicate through message passing (clusters). With this framework, they efficiently implement concurrent data structures much faster than their locking analogs.

Fast concurrent stack and queue implementations appear in [8, 9, 14, 25, 22, 27, 28, 29]. In [25], Morrison and Afek present a lock-free implementation of a concurrent queue, called LCRQ. LCRQ outperforms CC-Queue [9] and the queue based on flat-combining [14]. Our experiments show that OsciQueue outperforms LCRQ by a factor of more than 4 and PSimQueue-x outperforms LCRQ by a factor of more than 2. In [28], Tsigas and Zhang present a lock-free queue implementation which outperforms the lock-free queue of [22], as well as a blocking queue based on spin-locks; the queue is implemented as a circular array.

3 Model

We consider a system of m processing cores on which n threads $p_0, \ldots p_{n-1}$ (where n can be much larger than m) are executed. On each of the m cores, $t = n/m$ threads are executed (for simplicity, let $n \bmod m = 0$); one of these t threads is a kernel thread which spawns the other $t-1$ threads as user-level threads (by calling a function called `CreateThread`). We assume that thread p_i is executed on core i/t. Without loss of generality, we assume that thread $p_{j \cdot t}$ is the kernel thread of core j. We remark that the operating system may decide to move a kernel thread p (and all the user level threads that p has spawned) to another core. Notice also that it is only for the simplicity of the presentation that we make the above assumptions regarding the placement of the threads.

The kernel is aware only of the kernel threads and makes decisions about scheduling. If the kernel decides to switch context when one of these threads p has the CPU, p and all threads it has spawned, stop executing until the operating system decides to allocate the CPU to p again. A thread calls `Yield` whenever it wants to give the CPU. Then, a user-level scheduler, implemented by the corresponding kernel thread, is activated to decide which of the t threads of the core will next occupy the CPU. We assume that this choice is made in a *fair manner*.

We consider a *failure model*, where if a kernel thread p fails, then all threads executing in the same core also fail at the same point in time as p.

4 Description of Algorithms

Description of Osci. Osci (Algorithm 1) maintains a linked list of nodes (initially empty) for storing active requests. A shared pointer $Tail$, initially `NULL`, points to the last inserted node in the list. Each node v is of type ListNode and contains the requests announced by the active threads running on a single core c. These requests are recorded in the $reqs$ field of v, which is an array with as many elements as the number of threads running on c. A thread p_i, running on c, records its requests in the i mod t position of $reqs$ (we note that for cases where different number of threads run on each core, Osci still works).

One of the threads (let it be p) that have recorded requests in the head node of the list plays the role of the combiner. A combiner traverses the list and serves the requests recorded in each of the list nodes (lines 30-36), until either it has traversed all elements of the list or it has served h requests in total* (line 37). If any of these conditions holds, the combiner thread gives up its combining role by identifying one of the threads from those that have recorded their requests in the next node to be the new combiner (lines 43-44). We remark that at each point in time, if the list is non-empty, then there is exactly one combiner. On the other hand, if the list is empty, then no combiner exists.

Each thread owns (only) two nodes of type ListNode and uses them interchangeably to perform subsequent requests (lines 1 and 4), so the nodes are recycled in an efficient way. The first thread (let it be p_i) among those running on core $c = \lfloor i/t \rfloor$, that wants to apply a request, tries to store a pointer to one of its nodes (let this node be nd) in $Announce[c]$ (line 2) using `CAS`†. Notice that $Announce[c]$ may also be accessed simultaneously by other threads running on c. If p_i's `CAS` succeeds, p_i records a struct of type ThreadReq in $nd.reqs[i \bmod t]$. This struct contains a pointer req pointing to the request that p_i has initiated, the return value ret for this request and two boolean variables $completed$ and $locked$ (to which we refer as the $locked$ and $completed$ fields of p_i). If both $locked$ and $completed$ are equal to `false`, p_i becomes the new combiner. Whenever p_i performs spinning, it spins on its $locked$ field; if $locked$ becomes `false` while $completed$ is `true`, then a combiner has served p_i's request.

After p_i has recorded its request, it changes the $door$ field of nd from `LOCKED` (which was initially) to `OPEN` (line 9) and calls `Yield` (line 10) to allow other threads running on c to announce their requests in nd. We say that p_i is a *director* of core j. A director executes lines 4-18 and it is the only thread among those that run on the same core c that can later be a combiner (no other thread on c can be a combiner while applying this current batch of

* In order to prevent a combiner from traversing a continuously growing list, an upper bound h on the requests that p may serve is set; experiments show that h does not significantly affect performance; for our experiments, we have chosen $h = 2n$.

† `CAS` implementations on real machines usually return `true`/`false`. For simplicity of the presentation, we assume that `CAS` returns the old value of the memory location on which it is applied. We can trivially make the algorithm work with `CAS` instructions that return `true`/`false`.

Algorithm 1 Pseudocode for Osci.

```
constant t = ⌈n/m⌉, LOCKED= 0, OPEN= 1, CLOSE= 2;     // m is the number of cores, n is the number of threads
struct ThreadReq {                                    struct ListNode {
    ArgVal req;                                           ThreadReq reqs[0..t-1];
    RetVal ret;                                           int door;
    boolean completed, locked;    }                       struct ListNode *next;    }
shared ListNode *Tail = NULL, *Announce[0..m-1] = {NULL};
private ListNode node_i[0..1] = {⟨ {⟨ ⊥, ⊥, false, true⟩}, LOCKED, NULL⟩};
private boolean toggle_i = 0;

RetVal Osci(ArgVal arg) {                             // pseudocode for thread p_i, i ∈ {1, ..., n}
    ListNode *myNode, *tmpNode, *cur = NULL;
    int offset = i mod t, counter = 0, j;
1   myNode = &node_i[toggle_i];                       // choose one of the nodes of p_i to use
2   cur = CAS(Announce[⌊i/t⌋], NULL, myNode);
3   if (cur == NULL) {                                // p_i is the current director on core ⌊i/t⌋
4       toggle_i = 1 - toggle_i;
5       myNode→reqs[offset].req = arg;                // the director announces its request
6       myNode→reqs[offset].locked = true;
7       myNode→reqs[offset].completed = false;
8       myNode→next = NULL;
9       myNode→door = OPEN;
10      YIELD();                                      // pass control to some other thread
11      Announce[⌊i/t⌋] = NULL;
12      while (CAS(myNode→door, OPEN, CLOSE) != OPEN) YIELD();// close the door
13      tmpNode = SWAP(Tail, myNode);                 // append the node in the request list
14      if (tmpNode ≠ NULL) {                         // p_i is not the combiner
15          tmpNode→next = myNode;                    // set the next field of the previous node
16          while (myNode→reqs[offset].locked==true) YIELD();// p_i calls Yield() instead of spinning
17          if (myNode→reqs[offset].completed==true)  // check if p_i's request has already been applied
18              return myNode→reqs[offset].ret;
        }
19  } else {                                          // p_i is not the director
20      while (CAS(cur→door, OPEN, LOCKED) != OPEN)   // try to acquire the lock
21          if (cur→door == CLOSE) goto line 2;       // if door is closed start from scratch
22          else YIELD();
23      myNode = cur;
24      myNode→reqs[offset] = <arg, ⊥, false, true>; // p_i announces its request
25      myNode→door = OPEN;
26      while (myNode→reqs[offset].locked==true) YIELD();// yield to other threads of same core
27      if (myNode→reqs[offset].completed==true)      // check if p_i's request has already been applied
28          return myNode→reqs[offset].ret;
    }
29  tmpNode = myNode;                                 // p_i is the combiner
30  while(true) {
31      for (j = 0; j < t; j++) {                     // p_i applies the requests of threads executing on core j
32          if (tmpNode→reqs[j].completed == false) {
33              apply tmpNode→reqs[j].req and store the return value to tmpNode→reqs[j].ret;
34              tmpNode→reqs[j].completed = true;     // announce that tmpNode→ req[j] is applied
35              tmpNode→reqs[j].locked = false;       // unlock the corresponding spinning thread
36              counter = counter + 1;
            }
        }
37      if (tmpNode→next == NULL or counter ≥ h) break;// h is an upper bound of the combined requests
38      tmpNode = tmpNode→next;
    }
39  if (tmpNode→next == NULL) {                       // check if p_i's req is the only record in the list
40      if(CAS(Tail, tmpNode, NULL) == tmpNode)       // attempt to set Tail to NULL
41          return myNode→reqs[offset].ret;
42      while (tmpNode→next == NULL) YIELD();         // some thread has in the mean time appended a node
    }
43  for (j = 0; tmpNode→next→reqs[j].completed≠false; j++) noop;
44  tmpNode→next→reqs[j].locked = false;              // identify the new combiner
45  return myNode→reqs[offset].ret;
} // Osci
```

operations). In case that the director becomes a combiner, it also executes lines 29-45. To apply a request, a thread $p_j \neq p_i$ that is also executed on c, first checks whether the *door* of the node nd pointed to by Announce[c] is `OPEN`, and if this is so, it tries to *acquire the door* by setting the value of *door* to `LOCKED` using `CAS` (line 20). If the `CAS` succeeds, p_j records its request in nd (lines 23, 24), unlocks the *door* (line 25), and repeatedly calls `Yield` (line 26) until some combiner either serves its request (lines 32-36) or informs p_j to become the new combiner (line 44).

Since we assume fair scheduling of threads on each core, it is guaranteed that at some later point, p_i is re-activated and executes from line 11 on. Then, p_i changes the *door* field of nd to `CLOSE` (line 12) using `CAS` to avoid synchronization problems with other threads running on c that may simultaneously try to record their requests in nd. Next, p_i appends nd in the shared list by executing `Swap` (line 13). If p_i has appended its node in an empty list (i.e. if the condition of line 14 is `false`), or both the *locked* and *completed* fields in the entry of $nd \rightarrow reqs$ corresponding to p_i, are equal to `false` (lines 16, 17), p_i becomes a combiner.

If p_i does not become a combiner, it updates the *next* field of the previous node to point to its node (line 15), and repeatedly calls `Yield` (line 16) until a combiner either serves its request or informs p_i that it is the new combiner. In the first case, p_i returns on line 18, whereas in the second it executes the combiner code (lines 29-45).

If a thread p_i evaluates the if condition of line 39 to `true` but unsuccessfully executes the `CAS` on line 40, then some other thread p_j has succeeded in updating $Tail$ to point to a node nd' but it has not yet changed the *next* field of the node to which $Tail$ was previously pointed to point to nd'. Then, p_i has to wait until p_j sets *next* to point to nd' (line 42), to ensure that it will correctly choose the thread to become the next combiner (lines 43-44). Notice that on lines 43-44, p_i indicates the node with the smallest identifier among those that have recorded their requests in nd' as the new combiner. A combiner is the director of a core and owns the first node of the list that contains requests unserved by previous combiners.

Osci is linearizable. If a thread p executes the `CAS` of line 2 successfully (i.e. it becomes a director), and until the next time p_i is scheduled and executes line 12, no other thread on this core can become the director (since these threads will execute the `CAS` of line 2 unsuccessfully). Let α be any execution and consider a thread p_i that executes an instance A of Osci at some configuration C. Let $Tail(C)$ be the value of $Tail$ at C. For each i, $1 \leq i \leq n$, denote by $mynode_i(C)$ the value of variable $mynode$ at C. If A executes the `Swap` of line 13 and gets `NULL` as the response, denote by C_f the configuration resulting from the execution of line 13; in case there is at least one configuration C' in A such that $mynode_i(C') \rightarrow reqs[i \mod t].locked =$ `false` and $mynode_i(C') \rightarrow reqs[i \mod t].completed =$ `false`, denote by C_f the first of these configurations. We say that p_i is a *combiner* at C if C_f exists and C is a configuration that follows C_f in which A is active. We prove that in each configuration C, either $Tail(C)$ equals `NULL` and there is no combiner at C, or $Tail(C)$ points to a node nd and there is exactly one combiner at C. We also prove that only a combiner executes lines 30-45 implying that each request recorded in a list node is applied exactly once, as needed to prove linearizability.

Description of PSim and PSim-x. PSim uses (1) an array *Announce* of n entries, where each if the n threads announces its requests, (2) a bit vector *Toggles* which records the threads that have active requests, and (3) an `LL/SC` object $Tail$, which stores a pointer to the state of the object. Whenever a thread p_i wants to apply a request, it announces its request in $Announce[i]$. After that, it toggles $Toggles[i]$ (by executing a `Fetch&Add` instruction) to indicate that it has an active request. Thread p_i discovers which requests are active using

this vector and $Toggles$; p_i serves the active requests by executing their code on a local copy of the simulated state. Then, p_i executes an SC in an effort to change $Tail$ to point to this copy. These actions may have to be applied by p_i twice to ensure that its request has been served. Together with the simulated state, PSim stores a response value for each thread.

In PSim-x, each thread p calls `Yield` after announcing its request. This increases the combining degree of the algorithm. In most cases, when p is scheduled again, it finds out that its request has been completed, so it does not pay the overhead of executing the rest of the algorithm. These are the reasons for the better performance of PSim-x.

5 Performance Evaluation

We evaluated Osci and PSim-x in two different multiprocessor architectures. The first is a 64-core machine consisting of 4 AMD Opteron 6272 processors. Each of these processors consists of 2 dies and each die contains 4 modules, each of which contains 2 cores (64 logical cores). The second one is an Intel 40-core machine equipped with 4 Intel Xeon E5-4610 processors. Each processor consists of 10 cores, and each core executes two threads concurrently (80 logical cores). We used the gcc 4.8.5 compiler and the Hoard memory allocator [4] for all experiments. The operating system was Linux with kernel version 3.4. We performed a lot of experiments for different numbers of user level threads to achieve the best performance for each algorithm. All algorithms were carefully optimized and for those that use backoff schemes, we performed a lot of experiments to choose the best backoff parameters. To prohibit the linux scheduler from doing unnecessary kernel thread migrations, threads were bound in all experiments: the i-th thread was bound on core $\lfloor i/t \rfloor$, where $t = \lceil n/m \rceil$. Most of the commercially available shared memory machines provide `CAS` instructions rather than supporting `LL/SC`. For our experiments, we simulate an `LL` on some object O with a simple `read`, and an `SC` with a `CAS` on a timestamped version of O to avoid the ABA problem[‡].

For our experiments, we developed a library that provides basic support for user level threads. We note that our goal is not to present a new user-level threads library but to ensure that we use a library which is simple and provides fast context switching. By replacing our user-level threads library with any other library that ensures fast context switching, Osci and PSim-x would exhibit similar performance gains. The thread on library we used supports the operations `CreateThread`, `Join`, and `Yield`. A POSIX thread (*kernel-level* thread) runs on each core. This thread calls `CreateThread` to spawn the other *user-level* threads running on the same core. It calls `Join` to wait its children threads to complete. `Yield` activates a user-level scheduler which passes control to some other thread executing on the same core. `Yield` is implemented with a FIFO queue that stores the running threads on each processing core. Moreover, `Yield` makes the appropriate calls to standard *_setjmp/_longjmp* functions to context switch between user-level threads. For performance reasons, it is important that the implementation of `Yield` is as fast as possible and the scheduler is fair.

For our experiments, we first consider a synthetic benchmark, called the `Fetch&Multiply` benchmark, which is similar to that presented in [8, 9]. In this benchmark, a `Fetch&Multiply` object is simulated using state-of-the-art synchronization techniques such as CC-Synch [9], PSim [8], flat-combining [13, 14], a blocking implementation of a `Fetch&Multiply` object based on CLH queue spin-locks [7, 19], OyamaAlg [26], and a simple lock-free implementation. A `Fetch&Multiply` object supports the operation `Fetch&Multiply`(O, k) which returns

[‡] The ABA problem occurs when a thread p reads a value A from a shared variable O and then a thread p' modifies O to the value B and back to A; when p executes again, it thinks that the value of O has never changed which is incorrect.

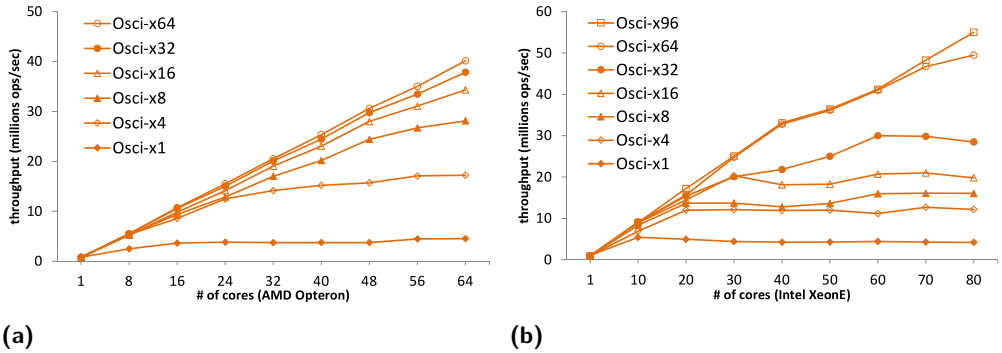
Table 2 Speedup of state-of-the-art synchronization algorithms when employing user level threads (AMD Opteron).

Algorithm	throughput	Variant	throughput	speedup
CC-Synch	4.18	CC-Synch-x8	4.60	1.10
DSM-Synch	4.10	DSM-Synch-x8	4.58	1.12
H-Synch	4.23	H-Synch-x32	10.1	2.39
PSim	3.90	PSim-x64	23.2	5.94
Lock-Free	2.00	Lock-Free-x2	1.87	0.94
CLH	1.58	CLH-x4	1.70	1.08
flat-combining	2.99	flat-combining-x24	5.51	1.84
OyamaAlg	1.72	OyamaAlg-x16	2.80	1.63

the current value v of memory location O and updates the value of O to be $v * k$. The considered lock free implementation uses a single `CAS` object. When a thread wants to apply a `Fetch&Multiply`, it repeatedly executes `CAS` until it succeeds; to increase the scalability, a backoff scheme is employed. The `Fetch&Multiply` object is simple enough to exhibit any overheads that a synchronization technique may induce while simulating a simple, small shared object. To avoid long runs and unrealistically low number of cache misses [22, 8], which may make the experiment unrealistic, we added some small local workload between two consecutive executions of `Fetch&Multiply` in a similar way as in [9, 22]. This local workload is implemented as a loop of dummy iterations whose number is chosen randomly (to be up to 512). For our machine configuration, this workload is large enough to avoid long runs and unrealistically low number of cache misses; still, it is small enough to allow large contention in the simulated object (see Figure 4b for more details). Each instance of the benchmark simulates 10^8 `Fetch&Multiply`, in total, with each of the n threads simulating $10^8/n$ `Fetch&Multiply` out of them. The average throughput is measured. Each experiment is executed 10 times and averages are taken.

In Table 2, we present the throughput for (1) the original versions of the evaluated algorithms (i.e. only one thread per core) and (2) their variants where many user level threads per core are created (for executing the `Fetch&Multiply` benchmark). The x$\langle yy \rangle$ suffix in their names indicates the number of user level threads that are executed per core, so CLH-x4 indicates that CLH spin locks are evaluated with 4 user level threads per core. We performed a lot of experiments for different numbers of user level threads in order to achieve the best performance for each algorithm. We also report the additional speedup we gain for each algorithm by using more than one user level thread per core. The performance of these variants appears in column 4 of the table. The performance presented in Table 2 was measured for the best number of user-level threads for each algorithm and it was performed on the AMD machine. For blocking algorithms that perform spinning on some shared variable, namely CC-Synch, flat-combining and the implementation based on CLH locks, we (repeatedly) call `Yield` instead of spinning. For implementations that repeatedly perform `CAS`, like OyamaAlg and the lock-free implementation, we call `Yield` between any two consecutive attempts to execute `CAS`.

Table 2 shows that all algorithms other than PSim, H-Synch, flat-combining and OyamaAlg do not exhibit any performance gain when employing user level threads. The main reason for this is that the total number of atomic primitives (i.e. `CAS`, `Swap` and `Fetch&Add`) that are executed by each of these algorithms is the same independently of how many user level threads are employed. We note that the performance of the simple lock-free implementation deteriorates for $n > m$, since then this variant (1) behaves similarly to the original algorithm



(a) (b)

Figure 2 Average throughput of Osci while simulating a `Fetch&Multiply` object on (a) the AMD machine (b) the Intel machine, for different numbers of user-level threads per core. The legends are listed top to bottom in the order of the curve they refer to.

(the identity of the thread that repeatedly attempts to execute `CAS` at each point in time is immaterial), and (2) has the additional cost of executing `Yield`.

It is noticeable that the performance of PSim improves by a factor of up to 5.9 when using user-level threads. This performance gain is due to the fact that each thread in PSim-x first announces its request, then calls `Yield` to allow to other threads on the same core to announce their requests, and finally checks if its request has been applied. Only if this is not so, PSim-x tries to serve all the pending requests. However, it turns out that in most of the cases, this is not needed, so the request is completed without paying the overhead of executing the combining part. This results in a big performance advantage of PSim-x. The version of flat combining that uses user level threads increases its performance by a factor of up to 1.84, while the performance of OyamaAlg increases by a factor of up to 1.63. However, PSim-x is 4.21 times faster than flat-combing, 8.2 times faster than OyamaAlg, and 3.8 times faster than H-Synch. We remark that the maximum performance for flat-combining is achieved for 24 user-level threads per core. Notice that H-Synch performs better than that of CC-Synch and PSim but much lower than that of Osci and PSim-x.

Figures 2a and 2b show Osci's performance for different numbers of user level threads per core when executing the `Fetch&Multiply` benchmark on the AMD and the Intel architectures, respectively. The x-axis of the diagram represent the number of cores, and the y-axis represents the average throughput of Osci (over the 10 runs performed). Each line of the diagrams corresponds to a different number of user-level threads per core. The local work between two consecutive `Fetch&Multiply` is up to 512 dummy iterations. The first figure shows that the performance of Osci increases as the number of user level threads increases. For small numbers of user level threads per core (up to 8), the performance gain is significant. Specifically, by using 4 user level threads per core, the performance of Osci is increased almost 4x, and when 8 user level threads per core are used, the performance increases by a factor of more than 6. However, smaller performance gains are illustrated in case of 16 or more user level threads since then the dominant performance factor becomes the switching overhead that `Yield` induces. Moreover, no significant improvement on Osci's performance is achieved when more than 32 threads are employed per core. Our experiments show that for big numbers of threads per core (e.g. more than 64), the performance of Osci slightly deteriorates. In the case of the Intel architecture (Figure 2b), the performance behavior of Osci is very similar, since its performance increases as the number of user level threads increases. The best performance for Osci is achieved for either 64 or 96 threads per core.

Figures 3a, 3b compare the performance of Osci and PSim-x with the other synchronization

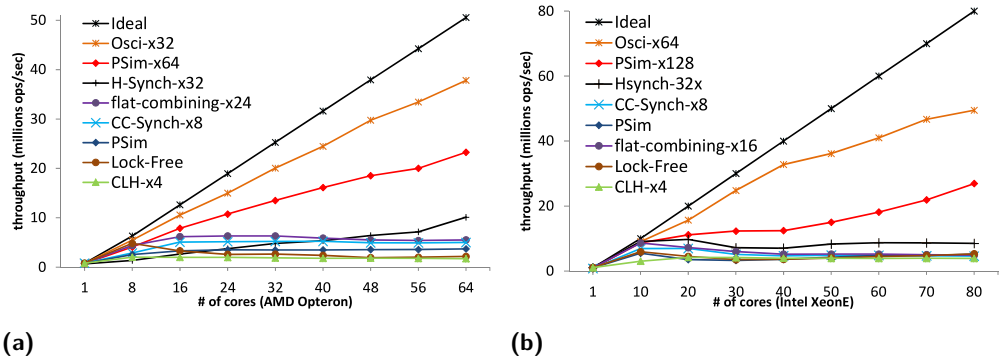


Figure 3 Average throughput of Osci and PSim-x against other synchronization techniques on (a) the AMD machine, and (b) the Intel machine.

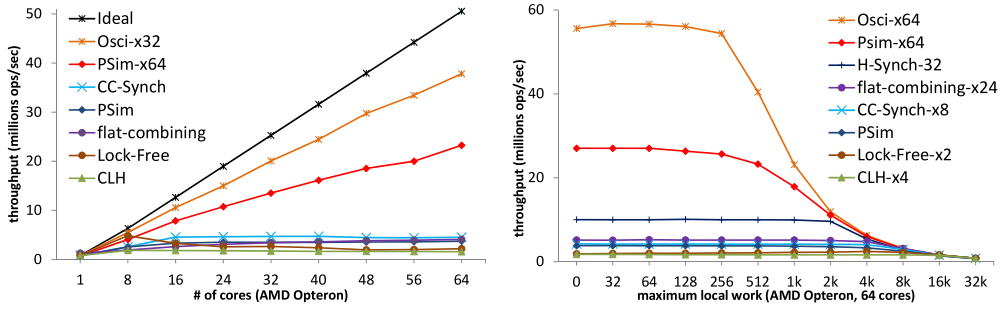
techniques (the versions that use more than one thread per core) running the `Fetch&Multiply` benchmark on the AMD and Intel machines, respectively. Driven by the results presented on Figures 2a, 2b, we configured Osci to use 32 user-level threads per core on the AMD machine and 64 on the Intel machine. Experiments showed that PSim-x scales well with 64 threads per core on the AMD machine and with 128 threads per core on Intel. Figures 3a, 3b also show the performance of an ideal implementation of the `Fetch&Multiply`. The throughput of the ideal implementation is calculated by multiplying n with the throughput of the sequential execution. This is higher than what the ideal algorithm would achieve (so this comparison is not in favor of our algorithms). For $n = 1$, the ideal performance is almost the same as the performance of Osci and CC-Synch [9] (and close to all other algorithms), since (1) the dominant performance factor is the local work, (2) there is no contention, and (3) all algorithms perform just 1 or 2 atomic primitives per request without causing cache misses. Notice that the performance of the ideal algorithm scales linearly to the number of cores.

Figure 3a shows that Osci and PSim-x outperform all other synchronization techniques (the versions that do not use more than one thread per core) by far. On the AMD machine, Osci outperforms CC-Synch [9] by a factor up to 7.5 and PSim-x outperforms CC-Synch by a factor up to 4.6. Recall that, Table 2 shows that the performance of CC-Synch, as well as of all other algorithms, do not improve much when using more than one user-level thread per core. This proves that the batching technique impacts performance significantly. Figure 3a also shows that Osci is up to 6.8 times faster than the variant of flat-combining that employs user-level threads and it is only up to 25% slower than the ideal. As expected [9], CC-Synch is faster than PSim and that the simple lock-free version of `Fetch&Multiply` performs similarly to CLH locks (on this benchmark). Figure 3b shows that the performance advantages of Osci and PSim-x on the Intel machine are very similar to those on the AMD machine. Due to lack of space, we focus our performance study of Osci and PSim-x on the AMD machine, from now on. The performance behavior of Osci and PSim-x on the Intel architecture is similar.

Figure 4a provides a comparison of the performance of Osci and PSim-x to that of the original algorithms (that employ just one thread per core). Similarly to Figure 3a, the performance advantages of Osci and PSim-x are significant. Table 3 shows that Osci does not significantly impact the average latency per operation. With 4 threads per core, the average latency is slightly increased (by less than 5% whereas its throughput is 3.81 times higher than that of CC-Synch. With 8 threads per core, the latency increases just 4 nsec (less than 29%) whereas the throughput is 6.23 times higher. Notice that for local work that is greater than $1k$, the dominant factor in system's performance is the local work and not the overhead of

Table 3 The impact of Osci in latency performance (random local work = 512).

	CC-Synch	Osci-x4	Osci-x8	Osci-x16	Osci-x32	H-Synch-x32
Average Latency (usec)	0.0142	0.0148	0.0182	0.0298	0.0541	0.0205
Throughput (millions ops/sec)	4.51	17.22	28.10	34.32	37.83	10.12



(a) (b)

Figure 4 (a) Average throughput of Osci and PSim-x against other synchronization techniques while simulating a `Fetch&Multiply` object on the AMD machine. (b) Average throughput of Osci and PSim-x with different values of random work for 64 cores.

the synchronization protocol (see Figure 4b). In this case the optimum performance should be achieved with low numbers of user-level threads per core leading to also low latency.

In Figure 4b, we evaluate the performance of Osci and PSim-x for different values of local random work on the AMD machine. In this benchmark, we use 64 user-level threads per core for Osci in order to achieve the best performance. It is shown that Osci and PSim-x outperform all the other synchronization, techniques (Osci outperforms CC-Synch and flat-combining by a factor of up to 11, respectively). Even in cases where the contention is low (local random work is equal to $4k$), Osci and PSim-x perform better (more than 1.5 times faster) than all other synchronization techniques. Smaller values of local work (and therefore higher contention) are in favor of Osci and PSim-x. For relatively large amounts of random local work, the experiment shows that the throughput starts to decline. For local work greater than $16k$, all synchronization techniques have similar performance, since then the local work becomes the dominant performance factor. This experiment shows that Osci and PSim-x would behave efficiently for a large collection of applications, since their performance is tolerant for different amounts of local work that the application may execute. Additionally, even with low contention, Osci and PSim-x perform better than all the other algorithms.

Table 4 sheds light on the reasons that Osci achieves good performance, providing the average number of cache misses (all levels) per request, the average number of cpu cycles spent in backend stalls per request, and the average combining degree achieved by each algorithm. Given that all the memory footprints of our benchmarks were small comparing to the processors' cache size, the great majority of cache misses are cache-line invalidations due to the coherency protocol. The number of cache misses and backend stalls was recorded with *perf* linux tool. As shown in Table 4, Osci exhibits the lowest amount of cache misses among all the other algorithms. Moreover, Osci causes the smallest number of cpu cycles spent in back-end stalls. Therefore, Osci not only executes the smallest amount of cache misses but also its cache-misses are the cheapest, since it spends the fewest cpu cycles in backend-stalls. This is due to the fact that Osci is able to announce an entire batch of active requests by executing just a single `Swap`. Similarly, the combiner reads (and applies) a batch of

Table 4 Last level cache misses and cpu cycles spent in backend stalls per request (64 cores, maximum random work = 64).

Algorithm	cache-misses (all levels)	cpu cycles spent in backend stalls	combining degree
Osci-x64	0.20	247.9	1404
PSim-x64	0.24	2306	1307
H-Synch-x32	0.47	666.1	32
CC-Synch	0.47	4210	1079
PSim	0.40	14300	22

requests without causing a cache miss for each request. We remark that all the other atomic primitives executed by a thread, performing an instance of Osci, access memory locations cached in the local core where the thread resides, avoiding invalidations on the caches of remote cores. PSim-x achieves the second lowest number of cache misses. Even though PSim-x's cache misses are more expensive than H-Synch-x32's (back-end stalls are more in this case), H-Synch-x32 spends a significant amount of cpu cycles spinning on a central lock. Also, the very low combining degree of H-Synch-x32 results in worse performance, comparing to PSim-x; for a bigger number of threads per core, H-Synch's performance deteriorates. PSim-x operates in a more complicated way than Osci in order to achieve wait-freedom. The main overhead of PSim-x compared to Osci is that, to execute a request, each thread locally copies the state of the simulated object and an array of return values of size $\Omega(n)$. Due to the over-subscription, the array of return values is pretty large in PSim-x. This results in a larger amount of cache-misses and backend stalls comparing to Osci. As PSim-x is a simple variant of PSim, the performance gains of PSim-x over PSim originate from the much better combing degree of PSim-x. This results in much less cache misses and cpu stalls.

6 Queue and Stack Implementations

We implement and experimentally analyze a shared queue, called OsciQueue, based on the two lock queue implementation by Michael and Scott [22]. In OsciQueue, the two locks of the lock queue [22] are replaced by two instances of Osci. We also study a version of SimQueue [8] that uses user level threads. This version is called PSimQueue-x.

In Figure 5a, we compare these queue implementations with state-of-the-art queue implementations, like the lock-free LCRQ implementation recently presented by Morrison and Afek in [25], the blocking CC-Queue implementation presented in [9], the wait-free SimQueue [8] implementation, and the lock free queue implementation presented by Michael and Scott in [22]. The experiment is performed on the AMD machine and it is similar to that presented in [22, 8, 9]. Specifically, each of the n threads executes $10^8/n$ *pairs* of ENQUEUE and DEQUEUE requests, starting from an empty data structure. This experiment is performed for different values of n. Similarly to the experiment of Figure 3a, a random local work (up to 512 dummy loop iterations) is simulated between the execution of two consecutive requests by the same thread. In the experiment of Figure 5a, the queue was initially empty. In our environment, OsciQueue achieves its best performance for 64 user-level threads per core. We use the LCRQ implementation that is provided in [24] and we performed a lot of experiments to determine the appropriate ring size of LCRQ that achieves the best performance.

Figure 5a shows that OsciQueue outperforms all other queue implementations by far. Specifically, OsciQueue is more than 4 times faster than LCRQ and more than 5 times faster

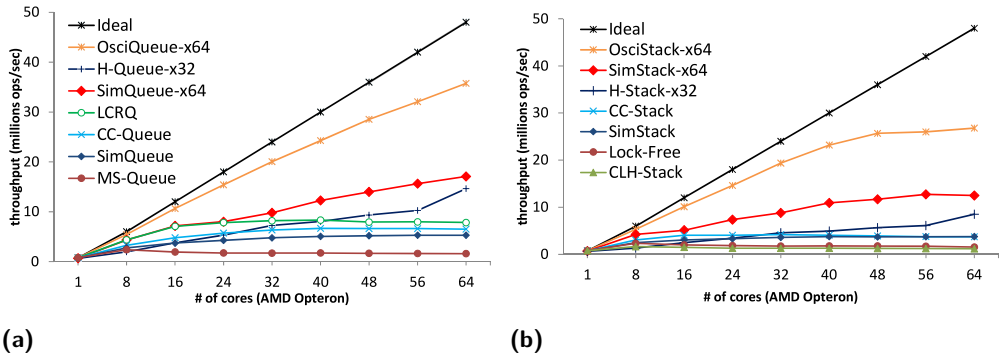


Figure 5 Average throughput of (a) OsciQueue, and (c) OsciStack.

than CC-Queue. It is also shown that PSimQueue-x outperforms LCRQ by a factor of 2. Notice that OsciQueue's performance is far from ideal by a factor of only 25%. Additionally, it ensures wait-freedom which is stronger than lock-freedom ensured by LCRQ. As it is expected [25], LCRQ has the best performance among all the other queue implementations. Recall that the queue is initially empty in this experiment. We also performed a similar experiment where the queue was initially containing 8192 elements. In this case, the performance results were very similar to those of Figure 5a.

Based on Osci and PSim-x, we derive implementations for concurrent stacks, called OsciStack and PSimStack-x, respectively. In Figure 5b, we compare their performance with the state-of-the-art shared stack implementations. More specifically, OsciStack and PSimStack-x were evaluated against CC-Stack [9], SimStack [8], the lock-free stack implementation presented by Treiber in [27], a stack implementation based on CLH spin locks [7, 19], where elimination has been applied when possible. The stack implementation recently presented in [6] is designed for a client-server model and thus it is not evaluated in this paper.

Similarly to the experiment of Figure 5a, we measure the average throughput that each algorithm achieves (every thread executes $10^8/n$ pairs of PUSH and POP requests) for different values of n. The random local work is set to 512. Figure 5b illustrates that OsciStack outperforms by far all other stack implementations. Specifically, OsciStack is up to 7.1 times faster than CC-Stack. It is noticeable that PSimStack-x, which is a wait-free stack implementation outperforms CC-Stack by a factor of up to 3.3.

7 Conclusions

In this paper a new combining technique, which is called Osci is presented. Osci shows remarkable performance when paired with cheap context switching. We have experimentally shown that Osci significantly outperforms all previous combining algorithms. Specifically, the throughput of Osci is higher than that of previously presented combining techniques by more than an order of magnitude. Notably, Osci's throughput is much closer to the ideal than all previous algorithms, while maintaining the average latency in serving each request low. Osci is evaluated in two different multiprocessor architectures, namely AMD and Intel.

Based on Osci, we have implemented and experimentally evaluated implementations of concurrent queues and stacks. These implementations outperform by far all current state-of-the-art concurrent queue and stack implementations. Although the current version of Osci has been evaluated in an environment supporting user-level threads, it would run correctly on any threading library, preemptive or not (including kernel threads).

References

1. Java threads in the solaris environment - earlier releases. URL: `http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/`.
2. User-mode scheduling in windows operating synstem. URL: `http://msdn.microsoft.com/en-us/library/windows/desktop/dd627187`.
3. Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.
4. Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2000.
5. Neil CC Brown. C++ csp2: A many-to-many threading model for multicore architectures. *Communicating Process Architectures 2007: WoTUG-30*, pages 183–205, 2007.
6. Irina Calciu, Justin E Gottschlich, and Maurice Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *Usenix Workshop on Hot Topics in Parallelism (HotPar)*, 2013.
7. T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Department of Computer Science, University of Washington, 1993.
8. Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, 2011.
9. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266. ACM, 2012.
10. Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.
11. J. R. Goodman, M.K. Vernon, and P. J. Woest. 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)*, pages 64–75, April 1989.
12. Panagiotis E Hadjidoukas and Vassilios V Dimakopoulos. Nested parallelism in the ompi openmp/c compiler. In *Euro-Par 2007 Parallel Processing*, pages 662–671. Springer, 2007.
13. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. The source code for flat-combining. URL: `http://github.com/mit-carbon/Flat-Combining`.
14. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 355–364, 2010.
15. Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
16. Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *7th International Conference on PGAS Programming Models*, 2013.
17. Randy H Katz, Susan J Eggers, David A Wood, CL Perkins, and Robert G Sheldon. *Implementing a cache consistency protocol*, volume 13. ACM, 1985.
18. David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Queue delegation locking. 2014.

19 Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 165–171, 1994.

20 Simon Marlow, S Peyton Jones, et al. The glasgow haskell compiler, 2004.

21 John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

22 Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

23 Seung-Hyun Min, Kwang-Ho Chun, Young-Rok Yang, and Myoung-Jun Kim. A soft real-time guaranteed java m: N thread mapping method. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1075–1080. Springer, 2005.

24 Adam Morrison and Yehuda Afek. The source code for LCRQ. . URL: `http://mcg.cs.tau.ac.il/projects/lcrq`.

25 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 103–112. ACM, 2013.

26 Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 182–204, 1999.

27 R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

28 Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA*, pages 134–143, 2001. `doi:10.1145/378580.378611`.

29 Philippas Tsigas, Yi Zhang, Daniel Cederman, and Tord Dellsen. Wait-free queue algorithms for the real-time java specification. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*, pages 373–383. IEEE Computer Society, 2006. `doi:10.1109/RTAS.2006.45`.

30 Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

31 Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *Computers, IEEE Transactions on*, 100(4):388–395, 1987.

Progress-Space Tradeoffs in Single-Writer Memory Implementations

Damien Imbs[*1], **Petr Kuznetsov**[*2], **and Thibault Rieutord**[*3]

1 **LIF, Aix-Marseille Université & CNRS, France, and Bremen University, Germany**
damien.imbs@lif.univ-mrs.fr
2 **LTCI, Télécom ParisTech, Université Paris Saclay, France**
petr.kuznetsov@telecom-paristech.fr
3 **LTCI, Télécom ParisTech, Université Paris Saclay, France**
thibault.rieutord@telecom-paristech.fr

Abstract

Many algorithms designed for shared-memory distributed systems assume the *single-writer multi-reader* (SWMR) setting where each process is provided with a unique register that can only be written by the process and read by all. In a system where computation is performed by a bounded number n of processes coming from a large (possibly unbounded) set of potential participants, the assumption of an SWMR memory is no longer reasonable. If only a bounded number of multi-writer multi-reader (MWMR) registers are provided, we cannot rely on an *a priori* assignment of processes to registers. In this setting, implementing an SWMR memory, or equivalently, ensuring *stable writes* (i.e., every written value persists in the memory), is desirable.

In this paper, we propose an SWMR implementation that adapts the number of MWMR registers used to the desired progress condition. For any given k from 1 to n, we present an algorithm that uses $n + k - 1$ registers to implement a *k-lock-free* SWMR memory. In the special case of 2-lock-freedom, we also give a matching lower bound of $n + 1$ registers, which supports our conjecture that the algorithm is space-optimal. Our lower bound holds for the strictly weaker progress condition of 2-obstruction-freedom, which suggests that the space complexity for k-obstruction-free and k-lock-free SWMR implementations might coincide.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation.

Keywords and phrases Single-writer memory implementation, comparison-based algorithms, space complexity, progress conditions

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.9

1 Introduction

We consider a model of distributed computing in which at most n *participating* processes communicate via reading and writing to a shared memory. The participating processes come from a possibly unbounded set of *potential* participants: each process has a unique identifier (IP address, RFID, MAC address, etc.) which we, without loss of generality, assume to be an integer value. Given that processes do not have an *a priori* knowledge of the participating set, it is natural to assume that they can only *compare* their identifiers to establish their relative order, otherwise they essentially run the same algorithm [14]. This model is therefore

* This work has been supported by the Franco-German DFG-ANR Project DISCMAT (14-CE35-0010-02) devoted to connections between mathematics and distributed computing.

called *comparison-based* [2]. In the comparison-based model with bounded shared memory, we cannot assume that the processes are provided with a prior assignment of processes to distinct registers. The only suitable assumption, as is the case for anonymous systems [16], is that processes have access to *multi-writer multi-reader* registers (MWMR), without a prior assignment.

In this paper, we study *space complexity* of comparison-based algorithms implementing an *abstract* single-writer multi-reader (SWMR) memory. The abstract SWMR memory allows each participating process to *write* to a private abstract memory location and to *read* from the abstract memory locations of all other participating processes. The SWMR abstraction can be further used to build higher-level abstractions, such as renaming [2] and atomic snapshot [1].

To implement an SWMR memory, we need to ensure that every write performed by a participating process on its abstract SWMR register is *persistent*: every future abstract read must see the written value, as long as it has not been replace by a more recent *persistent* write. To achieve persistence in a MWMR system, the emulated abstract write may have to update *multiple* base MWMR registers in order to ensure that its value is not overwritten by other processes. A natural question arises: *How many base MWMR registers do we need?*

In this paper, we show that the answer depends on the desired progress condition. It is immediate that n registers are required for a *lock-free* implementation, i.e., we want to ensure that at least *one* correct process makes progress. Indeed, any algorithm using $n-1$ or less registers can be brought into the situation where *every* base register is *covered*, i.e., a process is about to execute a write operation on it [4]. If we let the remaining process p_i complete a new abstract write operation, the other $n-1$ processes may destroy the written value by making a *block write* on the covered registers (each covering process performs its pending write operation). Thus, the value written by p_i is "lost": no future read would find it. It has been recently shown that n base registers are not only necessary, but also sufficient for a lock-free implementation [8].

A *wait-free* SWMR memory implementation that guarantees progress to *every* correct process can be achieved with $2n-1$ registers [8]. The two extremes, lock-freedom and wait-freedom, suggest an intriguing question: is there a dependency between the amount of progress the implementation provides and its space complexity: if processes are guaranteed more progress, do they need more base registers?

Contributions. In this paper, we give an evidence of such a dependency. Using novel covering-based arguments, we show that any 2-*obstruction-free* algorithm requires $n+1$ base MWMR registers. Recall that k-obstruction-freedom requires that every correct process makes progress under the condition that at most k processes are correct [15]. The stronger property of k-*lock-freedom* [5] additionally guarantees that if more than k processes are correct, then at least k out of them make progress.

We also provide, for any $k = 1, \ldots, n$, a k-lock-free SWMR memory implementation that uses only $n+k-1$ base registers. Our lower bound and the algorithm suggest the following:

▶ **Conjecture 1.** *It is impossible to implement a k-obstruction-free SWMR memory in the n-process comparison-based model using $n+k-2$ MWMR registers.*

An interesting implication of our results is that 2-lock-free and 2-obstruction-free SWMR implementations have the same optimal space complexity. Given that n-obstruction-freedom and n-lock-freedom coincide with wait-freedom, we expect that, for all $k = 1, \ldots, n$, k-obstruction-free and k-lock-free (and all progress conditions in between [5]) require the

same number $n + k - 1$ of base MWMR registers. Curiously, our results highlight a contrast between complexity and computability, as we know that certain problems, e.g., consensus, can be solved in an obstruction-free way, but not in a lock-free way [11].

Related work. There has been a lot of work on space complexity in distributed computing, e.g., [12, 10, 17].

Delporte et al. [9] studied the space complexity of anonymous k-set agreement using MWMR registers, and showed a dependency between space complexity and progress conditions. In particular, they provide a lower bound of $n - k + m$ MWMR registers to solve anonymous *repeated* k-set agreement in the m-obstruction-free way, for $k < m$. Delporte et al. [7] showed that obstruction-free k-set agreement can be solved in the n-process comparison-based model using $2(n - k) + 1$ registers. This upper bound was later improved to $n - k + m$ for the progress condition of m-obstruction-freedom ($m \leq k$) by Bouzid, Raynal and Sutra [3]. In particular, their algorithm uses less than n registers when $m < k$.

To our knowledge, the only lower bound on the space-complexity of implementing an SWMR memory has been given by Delporte et al. [8] who showed that lock-free comparison-based implementations require n registers.

Delporte et al. [8] proposed two SWMR memory implementations: a lock-free one, using n registers, and a wait-free one, using $2n - 1$ registers. These algorithms are used in [6] to implement a *uniform* SWMR memory, i.e., assuming no prior knowledge on the number of participating processes. Assuming that p processes participate, the algorithms use $3p + 1$ and $4p$ registers for, respectively, lock-freedom and wait-freedom.

Roadmap. The paper is organized as follows. Section 2 defines the system model and states the problem. Section 3 presents a k-lock-free SWMR memory implementation. Section 4 shows that a 2-obstruction-free SWMR memory implementation requires $n + 1$ MWMR registers and hence that our algorithm is optimal for $k = 2$. Section 5 concludes the paper with implications and open questions. Secondary proofs, similar to those from [8], are delegated to the appendix.

2 Model

We consider the asynchronous shared-memory model, in whicha bounded number $n > 1$ of asynchronous crash-prone processes communicate by applying read and write operations to a bounded number m of *base* atomic multi-writer multi-reader atomic registers. An atomic register i can be accessed with two memory operations: $write(i, v)$ that replaces the content of the register with value v, and $read(i)$ that returns its content. The processes are provided with unique identifiers from an unbounded name space. Without loss of generality, we assume that the name space is the set of positive integers.

2.1 States, configurations and executions

An algorithm assigned to each process is a (possibly non-deterministic) automaton that accepts high-level operation requests as an application input. In each state, the process is poised to perform a *step*, i.e., a read or write operations on base registers. Once the step is performed, the process changes its state according to the result the step operation, possibly non-deterministically and possibly to a step corresponding to another high-level operation.

A *configuration*, or system state, consists of the state of all processes and the content of all MWMR registers. In the *initial* configurations, all processes are in their initial states, and all registers carry initial values.

We say that a step e by a process p is *applicable* to a configuration C, if e is the pending step of p in C, and we denote by Ce the configuration reached from C after p performed e. A sequence of steps $e_1, e_2, \ldots$ is applicable to C, if e_1 is applicable to C, e_2 is applicable to Ce_1, etc. A (possibly infinite) sequence of steps applicable to a configuration C is called an *execution from* C. A configuration C is said to be *reachable* from a configuration C', and denoted $C \in \mathit{Reach}(C')$, if there exists a finite execution α applicable to C', such that $C = C'\alpha$. If omitted, the starting configuration is the initial configuration, and we write $C \in \mathit{Reach}()$ if C is a reachable configuration for our algorithm.

Processes that take at least one step of the algorithm are called *participating*. A process is called *correct* in a given (infinite) execution if it takes infinitely many steps in that execution. Let $\mathit{Correct}(\alpha)$ denote the set of correct processes in the execution α.

2.2 Comparison-based algorithms

We assume that the processes are allowed to use their identifiers only to compare them with the identifiers of other processes: the outputs of the algorithm only depend on the inputs, the relative order of the identifiers of the participating processes, and the schedule of their steps. Formally, we say that an algorithm is *comparison-based*, if, for each possible execution α, by replacing the identifiers of participating processes with new ones preserving their relative order, we obtain a valid execution of the algorithm. Notice that the assumption does not preclude using the identifiers in communication primitives, it only ensures that decisions taken in the algorithm's run are taken only based on the their relative order of the identifiers.

In this model, m MWMR registers can be used to implement a wait-free m-component *multi-writer atomic-snapshot* memory [1]. The memory exports operations $\mathit{Update}(i, v)$ (updating position i of the memory with value v) and $\mathit{Snapshot}()$ (atomically returning the contents of the memory). In the comparison-based atomic-snapshot implementation, easily derived from the original one [1], $\mathit{Update}(i, v)$ writes only once, to register i, and $\mathit{Snapshot}()$ is read-only. For convenience, in our upper-bound algorithm we are going to use atomic snapshots instead of read-write registers.

2.3 SWMR memory

A single-writer multi-reader (SWMR) memory exports two operations: $\mathit{Write}()$ that takes a value as a parameter and $\mathit{Collect}()$ that returns a *multi-set* of values. It is guaranteed that, in every execution, there exists a reading map π that associates each complete Collect operation C, returning a multi-set $V = \{v_1, \ldots, v_s\}$, with a set of s Write operations $\{w_1, \ldots, w_s\}$ performed, respectively, by distinct processes $p_1, \ldots, p_s$ such that:

- The set $\{p_1, \ldots, p_s\}$ contains all processes that completed at least one write operation before the invocation of C;
- For each $i = 1, \ldots, s$, w_i is either the last write operation of process p_i preceding the invocation of C or a write operation of p_i concurrent with C.

Note that our definition does not guarantee atomicity of SWMR operations. Moreover, we do not require that processes are allocated with a unique MWMR register that can be used as a single writer register. Instead, we simply require that processes are able to simulate the use of single writer registers through implementing the SWMR memory.

Intuitively, a collect operation can be seen as a sequence of reads on *regular* registers [13], each associated with a distinct participating process. Such a collect object can be easily transformed into a *single-writer* atomic snapshot abstraction [1].

2.4 Progress conditions

In this paper we focus on two families of progress conditions, both generalizing the *wait-free* progress condition, namely *k-lock-freedom* and *k-obstruction-freedom*.

An execution α satisfies the property of *k-lock-freedom* [5] ($1 \leq k \leq n$) if at least $\min(k, Correct(\alpha))$ correct processes *make progress* in it, i.e., complete infinitely many high-level operations (in our case, Writes and Collects). The special case of n-lock-freedom is called *wait-freedom*. The property of *k-obstruction-freedom* [11, 15] requires that every correct process makes progress, under the condition that there are at most k correct processes. (If more than k processes are correct, no progress is guaranteed.)

In particular, k-lock-freedom is a stronger requirement than k-obstruction-freedom (strictly stronger for $1 \leq k < n$). Indeed, both require that every correct process makes progress when there are at most k correct processes, but k-lock-freedom additionally requires that some progress is made even if there are more than k correct processes.

3 Upper bound: k-lock-free SWMR memory with $n + k - 1$ registers

Consider a *full-information* algorithm in which every process alternates atomic snapshots and updates, where each update performed by a process incorporates the result of its preceding snapshot. Every value written to a register will *persist* (i.e., will be present in the result of every subsequent snapshot), unless there is another process poised to write to that register. The pigeonhole principle implies that k processes can cover at most k distinct registers at the same time. Thus, if, at a given point of a run, a value is present in n registers, then the value will persist. This observation implies a simple n-register *lock-free* SWMR implementation in which a high-level Write operation alternates snapshots and updates of all registers, one by one in the round-robin fashion, until the written high-level value is present in all n registers. A high-level Collect operation can simply return the set of the most recent values (defined using monotonically growing sequence numbers) returned by a snapshot operation. 0+0 The *wait-free* SWMR memory implementation in [8] using $2n - 1$ registers follows the n-register lock-free algorithm but, intuitively, for each participating process, replaces register n with register $n - 1 + pos$ where pos is the rank of the process among the currently observed participants. This way, there is a time after which every participating process has a dedicated register to write, and each value it writes will persist. In particular, every value it writes will be seen by all processes and will eventually be propagated to the $n - 1$ first registers.

To implement a k-lock-free SWMR memory using $n + k - 1$ registers, a process should determine, in a dynamic fashion, to which out of the last $k - 1$ registers to write. In our algorithm, by default, a Write operation only uses the first n registers, but if a process observes that its value is absent from some registers in the snapshot (some of its previous writes have been overwritten by other processes), it uses extra registers to propagate its value. The number of these extra registers depends on how many other processes have been observed as making progress.

Algorithm 1: k-lock-free SWMR implementation using $n+k-1$ MWMR registers.

```
1  View : list of triples of type (ValueType, IdType, ℕ), initially set to ∅;
2  opCounter ∈ ℕ, initially set to 0;

3  Write(v):
4      ActiveProcs = {id};
5      View = View ∪ (v, id, opCounter);
6      WritePos = 0;
7      WritePosMax = n;
8      do
9          Snap = MEM.snapshot();
10         ActiveProcs = ActiveProcs ∪ {pid, ∃(_, pid, c) ∈ Snap, ∀(_, pid, c') ∈ View, c > c'};
11         View = View ∪ Snap;
12         Update(MEM[WritePos], View);
13         WritePos = WritePos + 1 (mod WritePosMax);
14         WritePosMax = min(n + |ActiveProcs| − 1, n + k − 1);
15     while |{m ∈ {1, ..., n + k − 1}, (v, id, opCounter) ∈ Snap[m]}| < n;
16     opCounter = opCounter + 1;
17 End Write;

18 Collect():
19     Reads = MEM.snapshot();
20     V = ∅;
21     forall pid such that (_, pid, _) ∈ Reads do
22         V = V ∪ {v} with v such that (v, pid, max{c ∈ ℕ, (_, pid, c) ∈ Reads}) ∈ Reads;
23     Return V;
24 End Collect;
```

3.1 Overview of the algorithm

Our k-lock-free SWMR implementation, which uses $n+k-1$ base MWMR registers, is presented in Algorithm 1.

In a Write operation, the process adds the operation to be performed to its local view (line 5). The process then attempts to add its local view, together with the outcome of a snapshot, to each of the first *WritePosMax* registers, where *WritePosMax* is initially set to n and then adapts to the number of processes observed as concurrently active (lines 8–15). The writing process continues to do so until its Write operation value is present in at least n registers (line 15).

In this algorithm, the $k-1$ extra registers are used according to the liveness observed by blocked processses. In order to be allowed to use the last register, a process must fail to complete its write while observing at least $k-1$ other processes completing their own. This ensures that when a process access this last register, a k^{th} process is able to be observed by processes completing operations and thus will be helped to eventually complete.

The Collect operation is rather straightforward. It simply takes a snapshot of the memory and, for each participating process observed in the memory, it returns its most recent value (selected using associated sequence numbers, line 22).

3.2 Safety

At a high level, the safety of Algorithm 1 relies on the following property of register content stability:

▶ **Lemma 2.** *Let, at some point of a run of the algorithm, value (v, id, c) be present in some register r and such that no process is poised to execute an update on r (i.e., no process is between taking the snapshot of MEM (line 9) and the update of r (line 12)), then at all subsequent times $(v, id, c) \in r$, i.e., the value is present in the set of values stored in r.*

The persistence of the values in a specific uncovered register (Lemma 2) can be used to show the persistence of the value of a completed Write operation in *MEM*:

▶ **Lemma 3.** *If process p returns from a Write operation $(v, id(p), c)$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(p), c)$.*

The proofs of the two above lemmas follow the path proposed in [8] and are delegated to the appendix.

With Lemma 3, we can derive the safety of our SWMR memory implementation (Section 2.3):

▶ **Theorem 4.** *Algorithm 1 safely implements an SWMR memory.*

Proof. It can be easily observed that a triplet (v, id, c) corresponds to a unique Write operation of a value v, performed by the process with identifier id. Therefore, a Collect operation returns a set of values proposed by Write operations from distinct processes, and thus the map π is well-defined.

By Lemma 3, the value (v, id, c) corresponding to a Write operation completed at time τ is present in some register r for any time $\tau' > \tau$, thus, the set of values resulting from any snapshot operation performed after time τ contains (v, id, c). Thus, for any complete Collect operation C, $\pi(C)$ contains a value for every process which completed a Write operation before C was invoked. Also, as each value returned by a Collect is the one associated to the greatest sequence number for a given process, it comes from the last completed Write or from a concurrent one. ◀

3.3 Progress

We will show, by induction on k, that Algorithm 1 satisfies k-lock-freedom. We first show, as in [8], that Write operations of Algorithm 1 are 1-lock-free (the proof is delegated to the appendix):

▶ **Lemma 5.** *Write operations in Algorithm 1 satisfy 1-lock-freedom.*

The induction step relies primarily on the helping mechanism. This mechanism guarantees that a process making progress eventually ensures that the processes it observes as having a pending operation also make progress (the mechanism is similar to the one of the wait-free implementation of [8]; the proof is also delegated to the appendix):

▶ **Lemma 6.** *If a process q performing infinitely many operations sees $(v, id(p), c)$, and if p is correct, then p eventually completes its c^{th} Write operation.*

By the base case provided by Lemma 5 and Lemma 6, we have:

▶ **Lemma 7.** *Write operations in Algorithm 1 satisfy k-lock-freedom.*

Proof. We proceed by induction on k, starting with the base case of $k = 1$ (Lemma 5). Suppose that Write operations satisfy ℓ-lock-freedom for some $\ell < k$. Consider a run in which at least $\ell + 1$ processes are correct, but only ℓ of them make progress (if such a run doesn't exist, the algorithm satisfies $(\ell + 1)$-lock-freedom). In this run, at least one correct process is eventually blocked in a Write operation. According to Lemma 6, the ℓ processes performing infinitely many Write operations eventually do not observe new values written by other processes. By the algorithm, these processes eventually never write to the last $k - \ell > 0$ registers.

A correct process that never completes a Write operation will execute the while loop (lines 8–15) infinitely many times, and thus, will infinitely often take a snapshot and update its local view (line 9). In particular, it will eventually observe a new Write operation performed by each of the ℓ processes completing infinitely many Write operations. It will then eventually include at least $\ell + 1$ processes in its set of active processes (i.e., the ℓ processes performing infinitely many Write operations and itself). It will therefore eventually write to the $(n + \ell)^{th}$ register infinitely often. In the considered run, this register is written infinitely often only by correct processes which do not complete new Write operations. The value from at least one of such process will then be observed by the ℓ processes making progress. By Lemma 6, this process will eventually complete its Write operation — a contradiction. ◀

Collect operations in Algorithm 1 clearly satisfy wait-freedom as there are no loops and MWMR snapshot operations are wait-free. Thus Lemma 7 and the wait-freedom of Collect operations imply:

▶ **Theorem 8.** *Algorithm 1 is a k-lock-free implementation of an SWMR memory for n processes using $n + k - 1$ MWMR registers.*

4 Lower bound: impossibility of 2-obstruction-free SWMR memory implementations with n MWMR registers

The algorithm in Section 3 gives an upper bound of $n+k-1$ on the number of MWMR registers required to implement an SWMR memory satisfying the k-lock-free progress condition in the comparison-based model. In this section, we present a lower bound on the number of MWMR registers required in order to provide a 2-obstruction-free, and hence also a 2-lock-free, SWMR memory implementation.

4.1 Overview of the lower bound

Our proof relies on the concepts of *covering* and *indistinguishability*.

A register is *covered* at a given point of a run if there is at least one process poised to write to it (we say that the process covers the register). Hence, a covered register cannot be used to ensure persistence of written data: by awakening the covering process, the adversarial scheduler can overwrite it. This property alone can be used to show that n registers are required for an obstruction-free (and hence also for a 1-lock-free) SWMR memory implementation [4], but not to obtain a lower bound of *more* than n shared resources as there is always one which remains uncovered.

Indistinguishability captures bounds on the knowledge that a process has of the rest of the system. Two system states are *indistinguishable* for a process if it has the same local state in both states and if the shared memory includes the same content. Thus, in an SWMR memory implementation, a Write operation can safely terminate only if, in all indistinguishable states, its value is present in a register that is not covered (by a process unaware of that value).

In our proof, we work with a composed notion of covering and indistinguishability. The idea is to show that there is a large set of reachable system states, indistinguishable to a given process p, in which different *sets of registers* are covered. Intuitively, if a set of registers is covered in one of these indistinguishable states, p must necessarily write to a register outside of this set in order to complete a new Write operation. Hence, if such indistinguishable states exist for *all* register subsets, then p must write its value to *all* registers. To perform infinitely many high-level Write operations, p must then write infinitely often to all available registers. But then any other process p' taking steps can be masked by the execution of p (i.e., any write p' makes to a MWMR register can be scheduled to be overwritten by p). This way we establish that no 2-obstruction free implementation exists, as it requires that at least two processes must be able to make progress concurrently.

4.2 Preliminaries

Assume, by contradiction, that there exists a 2-obstruction-free SWMR implementation using only n registers. To establish a contradiction, we consider a set of runs by a fixed set Π of n processes in which every process performs infinitely many Write operations with monotonically increasing arguments. Let $\mathcal{R}$ denote the set of n available registers.

Indistinguishability

A configuration C is said to be *indistinguishable* from a configuration C' for a set of processes P, if the content of all registers and the states of all processes in P are identical in C and C'. Given a set of configurations $\mathcal{D}$, let $I(\mathcal{D}, P)$ denote that any two configurations from $\mathcal{D}$ are indistinguishable for P.

We say that an execution is P-only, for a set of processes P, if it consists only of steps by processes in P. We say that a set of processes P is *hidden* in an execution α if all writes in α performed by processes in P are overwritten by some processes not in P, without any read performed by processes not in P in between. Given a sequence of steps α and a set of processes P, let $\alpha|_P$ be the sub-sequence of α containing only the steps from processes in P.

▶ **Observation 9.** *If a P-only execution α is applicable to a configuration C from a set of configurations $\mathcal{D}$ indistinguishable for P, i.e., $C \in \mathcal{D}$ and $I(\mathcal{D}, P)$, then α is applicable to any configuration $C' \in \mathcal{D}$, and it maintains the indistinguishability of configurations for P, i.e., $I(\{C'\alpha, C' \in \mathcal{D}\}, P)$.*

Let us denote as $\mathcal{D}\alpha$ the set of all configurations reached from $C \in \mathcal{D}$ by applying α. A similar observation can be made concerning hidden executions:

▶ **Observation 10.** *Given an execution α applicable to C with C from a set of configurations $\mathcal{D}$ indistinguishable for P, if processes in $\Pi \setminus P$ are hidden in α, then $\alpha|_P$ is applicable to any $C' \in \mathcal{D}$, and $I(\{C'\alpha|_P, C' \in \mathcal{D}\}, P)$.*

Coverings and confusion

We say that a set of processes P *covers* a set of registers R in some configuration C, if for each register $r \in R$, there is a process $p \in P$ such that the next step of p in C is a write on r (the predicate is denoted $Cover(R, P, C)$).

Our lower bound result relies on a concept that we call *confusion*. We say that a set of processes P are *confused* on a set of registers S in a set of reachable configurations $\mathcal{D}$, denoted $Confused(P, S, \mathcal{D})$, if and only if:

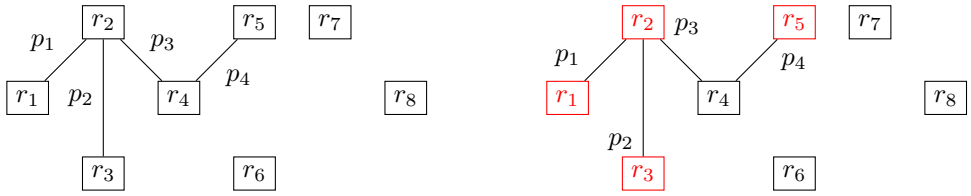


Figure 1 Processes $\{p_5, p_6, p_7, p_8\}$ are confused on registers $\{r_1, r_2, r_3, r_4, r_5\}$; an example of a possible covering is given on the right.

1. $I(\mathcal{D}, P)$.
2. $|S| + |P| = n + 1$.
3. For any process $p \in \Pi \setminus P$, there exist two registers $r_p, r'_p \in S$ such that, for any configuration $D \in \mathcal{D}$, there exists $D' \in \mathcal{D}$, such that p covers r_p in D and r'_p in D', or vice versa, and D and D' are indistinguishable to all other processes:

   $$\forall p \in \Pi \setminus P, \exists r_p, r'_p \in S, \forall D \in \mathcal{D}, \exists r \in \{r_p, r'_p\} :$$

   $$\mathit{Cover}(\{r\}, \{p\}, D) \wedge (\exists D' \in \mathcal{D}, I(\{D, D'\}, \Pi \setminus \{p\}) \wedge \mathit{Cover}(\{r_p, r'_p\} \setminus \{r\}, \{p\}, D')).$$

4. For any strict subset R of S, there exists $D \in \mathcal{D}$ such that R is covered by $\Pi \setminus P$ in D:

   $$\forall R \subsetneq S, \exists D \in \mathcal{D} : \mathit{Cover}(R, \Pi \setminus P, D).$$

Intuitively, processes in P are confused on S in $\mathcal{D}$, if $\mathcal{D}$ is a set of indistinguishable configurations for P, such that any *strict* subset of S is covered by $\Pi \setminus P$ in some configuration of $\mathcal{D}$ (Conditions 1 and 4). Additionally, we require that as much processes are confused as possible (Condition 2) and that the property holds for a set of configurations $\mathcal{D}$ where processes not in P might be covering only one out of 2 given registers and may be covering them independently of other processes states in $\mathcal{D}$ (Condition 3).

In Figure 1, we give an example of confusing set of configuration $\mathcal{D}$ for 8 processes and 8 registers. Processes $\{p_5, p_6, p_7, p_8\}$ are confused on registers $\{r_1, r_2, r_3, r_4, r_5\}$. Registers are represented as nodes, and pairs of registers that a process might be covering are represented as edges. The set of indistinguishable configurations $\mathcal{D}$ for $\{p_5, p_6, p_7, p_8\}$ are defined via composition of states for p_1, p_2, p_3 and p_4 in which they, respectively, cover registers in $\{r_1, r_2\}$, $\{r_2, r_3\}$, $\{r_2, r_4\}$ and $\{r_4, r_5\}$. An example of a covering of $\{r_1, r_2, r_3, r_5\}$ for some particular execution is presented on the right side of Figure 1.

First, we are going to provide an alternative property for Condition 4 of the definition of $\mathit{Confused}(P, S, \mathcal{D})$. The idea is that, given $(P, S, \mathcal{D})$ satisfying Conditions 1, 2 and 3, Condition 4 is satisfied if and only if the graph induced by the sets of registers that may be covered by processes in $\Pi \setminus P$ (as represented in Figure 1) forms a connected component over S. More formally, that Condition 4 is satisfied if and only if, for any partition of S into two non-empty subsets S_1 and S_2, there is a process in $\Pi \setminus P$ for which the set of two registers it may be covering in $\mathcal{D}$ intersects with both S_1 and S_2:

▶ **Lemma 11.** *$\forall P \subseteq \Pi, \forall S \subseteq \mathcal{R}, \forall \mathcal{D} \subseteq \mathit{Reach}()$ satisfying Conditions 1, 2 and 3 of the confusion definition, we have $\forall R \subsetneq S, \exists D \in \mathcal{D} : \mathit{Cover}(R, \Pi \setminus P, D)$ if and only if:*

$$\forall S_1, S_2 \subseteq S, (S_1 \neq \emptyset \wedge S_2 \neq \emptyset \wedge S_1 \cup S_2 = S \wedge S_1 \cap S_2 = \emptyset) :$$

$$\exists r_1 \in S_1, r_2 \in S_2, p \in \Pi \setminus P, D_1, D_2 \in \mathcal{D} : (\mathit{Cover}(\{r_1\}, \{p\}, D_1) \wedge \mathit{Cover}\left((\{r_2\}, \{p\}, D_2)\right).$$

Proof. Let us fix some $P \subseteq \Pi$, $S \subseteq \mathcal{R}$, and $\mathcal{D} \subseteq \mathit{Reach}()$ satisfying Conditions 1, 2 and 3 of the confusion definition.

First, let us assume that Condition 4 is also satisfied and consider any partition of S into non-empty subsets S_1 and S_2 (i.e., $S_1 \neq \emptyset$, $S_2 \neq \emptyset$, $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$). Assume now that there does not exist any process $p \in \Pi \setminus P$ such that p might be covering a register from S_1 or a register from S_2 in $\mathcal{D}$. This implies that processes in $\Pi \setminus P$ can be partitioned into two subsets Q_1 and Q_2 (with $Q_1 \cap Q_2 = \emptyset$ and $Q_1 \cup Q_2 = \Pi \setminus P$) such that processes in Q_1, respectively Q_2, may cover registers from S_1, respectively S_2, in $\mathcal{D}$. By construction of the partitions, we have $|Q_1| + |Q_2| = |\Pi \setminus P|$ and $|S_1| + |S_2| = |S|$. Using the fact that Condition 2 is satisfied by P and S we obtain from $|S| + |P| = n + 1$ that $|S_1| + |S_2| + (n - (|Q_1| + |Q_2|)) = n + 1$, and thus, that $|S_1| + |S_2| = |Q_1| + |Q_2| + 1$. This implies that either $|Q_1| < |S_1|$ or $|Q_2| < |S_2|$, w.l.o.g., let $|Q_1| < |R_1|$. Now consider $r \in S_2$, $S \setminus \{r\}$ is a strict subset of S, and therefore Condition 4 implies that there exists $D \in \mathcal{D}$ such that $\mathit{Cover}(S \setminus \{r\}, \Pi \setminus P, D)$. As registers in S_1 can only be covered by processes from Q_1, then we have $\mathit{Cover}(S_1, Q_1, D)$. Recall that, by the pigeonhole principle, a set of processes cannot cover more registers than processes it contains. But $|Q_1| < |R_1|$ — a contradiction.

Now let us assume that given any partition of S into non-empty subsets S_1 and S_2, there exists a process $p \in \Pi \setminus P$ such that p might be covering a register in S_1 or a register in S_2 in $\mathcal{D}$. Let us show that any strict subset R of S is covered in some configuration from $\mathcal{D}$ and, hence, that Condition 4 is satisfied. The idea consists in inductively selecting a subset of the configurations in $\mathcal{D}$ by fixing a process in $\Pi \setminus P$ to cover a new register of R. The trick is to select a process which remains with a single choice if it wants to cover a register not yet covered in all remaining configurations by other processes.

Let S_0 be a non-empty subset of S. Let p_0 be a process from $\Pi \setminus P$ which might be covering a register r_0 in S_0 or a register r'_0 in $S \setminus S_0$ in $\mathcal{D}$. Let us assume that such a process exists and consider $\mathcal{D}_0$ to be the subset of $\mathcal{D}$ including all configurations in which p_0 is covering r'_0. Now let $S_1 = S_0 \cup \{r'_0\}$ and repeat this process by selecting some p_1 and computing $\mathcal{D}_1$ using the same procedure, etc... As long as a process can be selected satisfying the condition, the sets S_i keeps increasing with i. Consider the round j at which the procedure fails to find such a process. This implies that there is no process which might be covering a register from either S_j or $S \setminus S_j$ in $\mathcal{D}_{j-1}$. Note that by construction $\mathcal{D}_{j-1}$ is a non-empty subset of $\mathcal{D}$.

If $S_j \neq S$, then S_j and $S \setminus S_j$ forms a partition of S into two non-empty subsets. Thus, by assumption, there exists a process q which might be covering a register r_q in S_j or a register r'_q in $S \setminus S_j$ in $\mathcal{D}$. Consider some configuration $D \in \mathcal{D}_{j-1}$. According to Condition 3 of the confusion definition, as $D \in \mathcal{D}$, q is covering either r_q or r'_q in D, and there exists a configuration D' in which q is covering the other register in $\{r_q, r'_q\}$, relatively to D, and such that D and D' are indistinguishable to all other processes. As in D and D', all processes except q are in the same local state, either $D' \in \mathcal{D}_{j-1}$ or else q was a process selected in some early iteration. But if q was selected in an earlier iteration, both r_q and r'_q would be included in S_j. Thus D and D' belong to $\mathcal{D}_{j-1}$ and therefore q is a valid selection for p_j. This contradiction implies that therefore $S_j = S$.

By construction, all registers in $S_j \setminus S_0$ are covered in all configurations in $\mathcal{D}_{j-1}$. As this is true for any non-empty S_0 and as $S_j = S$, any strict subset of S is covered in some configuration of $\mathcal{D}$ and therefore Condition 4 is satisfied. ◀

The alternative formulation of Condition 4 provided by Lemma 11 can be used to show that, given any confusion for some non-empty P, S and $\mathcal{D}$, we can identify a process $p \in \Pi \setminus P$ and a register $r \in S$ such that $P \cup \{p\}$ is confused on $S \setminus \{r\}$ for a subset $\mathcal{D}'$ of $\mathcal{D}$:

▶ **Lemma 12.** *Given $P \subsetneq \Pi$, $S \subseteq \mathcal{R}$ and $\mathcal{D} \subseteq Reach()$:*
$Confused(P, S, \mathcal{D}) \implies (\exists p \in \Pi \setminus P, \exists r \in S, \exists \mathcal{D}' \subseteq \mathcal{D} : Confused(P \cup \{p\}, S \setminus \{r\}, \mathcal{D}'))$.

Proof. Note that, according to Condition 3, a process may be covering exactly two registers from S in $\mathcal{D}$, thus, the sum of how many distinct processes may be covering each register in S equals $2|\Pi \setminus P| = 2(n - |P|)$. But any register $r \in S$ must be potentially covered at least by one process in $\Pi \setminus P$ in $\mathcal{D}$ as any strict subset of S may be covered (Condition 4). Therefore, there exists a register $r_c \in S$ which can be covered by a single process $p_c \in \Pi \setminus P$ in $\mathcal{D}$. Indeed, if all registers in S might be covered by two distinct processes then the sum, equal to $2(n - |P|)$, would be greater than or equal to $2|S|$. This is not possible as according to Condition 2, $|P| + |S| = n + 1$ and thus $n - |P| < |S|$.

Let us fix some $D_c \in \mathcal{D}$ and let $\mathcal{D}_c$ be the subset of $\mathcal{D}$ which includes all configurations in $\mathcal{D}$ that are indistinguishable to p_c from D_c. Let us show that $Confused(P \cup \{p_c\}, S \setminus \{r_c\}, \mathcal{D}_c)$. Condition 1 holds as by construction all configurations in $\mathcal{D}_c$ are indistinguishable to p_c and as they are indistinguishable to all processes in P, since $\mathcal{D}_c$ is a subset of $\mathcal{D}$. It is immediate, as we add a register and remove a process, that Condition 2 holds.

Now consider any process $p \in \Pi \setminus (P \cup \{p_c\})$ and any configuration $D \in \mathcal{D}_c$. As $p \in \Pi \setminus P$ and $D \in \mathcal{D}$, Condition 3 of the confusion definition implies that that there exists $D' \in \mathcal{D}$, $I(\{D, D'\}, \Pi \setminus \{p\})$, such that p covers r_p and r'_p in D and D' (respectively of vice-versa). Since $I(\{D, D'\}, \Pi \setminus \{p\})$, $D' \in \mathcal{D}_c$, and since p_c is the only process which may cover r_c in $\mathcal{D}$, r_p and r'_p belong to $S \setminus \{r_c\}$. Thus Condition 3 is verified for $P \cup \{p\}$, $S \setminus \{r_c\}$ and $\mathcal{D}_c$.

Lastly, let us consider some partition of $S \setminus \{r_c\}$ into two non-empty subsets S_1 and S_2. Both $(S_1 \cup \{r_c\}, S_2)$ and $(S_1, S_2 \cup \{r_c\})$ form a partition of S in two non-empty subsets. Thus, as $Confused(P, S, \mathcal{D})$, we can apply Lemma 11 and obtain that $\exists p_1, p_2 \in \Pi \setminus P$ such that p_1, respectively p_2, might cover registers from either $S_1 \cup \{r_c\}$ or S_2, respectively either S_1 or $S_2 \cup \{r_c\}$, in $\mathcal{D}$. It follows that p_c cannot be both p_1 and p_2 as p_c might cover only two registers in $\mathcal{D}$, one of which is r_c. Thus, depending whether the other register belongs to S_1 or S_2, p_1 or p_2 is disctinct from p_c. W.l.o.g, assume that it is p_1. As p_c is the only process which may be covering r_c, this implies that p_1 might be covering a register from either S_1 or S_2. Furthermore, since Conditions 1, 2 and 3 applies to $P \cup \{p\}$, $S \setminus \{r_c\}$ and $\mathcal{D}_c$, we can apply Lemma 11 to obtain that Condition 4 is also verified. ◀

We now show that the characterization can be used to increase the number of registers that processes are confused on, by decreasing the number of confused processes:

▶ **Lemma 13.** *Let $P \subsetneq \Pi$, $S \subseteq \mathcal{R}$ and $\mathcal{D} \subset Reach()$ such that $Confused(P, S, \mathcal{D})$.*
Given $C \in \mathcal{D}$, if $\exists p \in P, r_1 \in S, r_2 \in \mathcal{R} \setminus S$ and if there exists P-only executions α_1 and α_2, applicable to C such that $I(\{C\alpha_1, C\alpha_2\}, \Pi \setminus \{p\})$ and such that $Cover(\{r_1\}, \{p\}, C\alpha_1)$ and $Cover(\{r_2\}, \{p\}, C\alpha_2)$, then we have $Confused(P \setminus \{p\}, S \cup \{r_2\}, (\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2))$.

Proof. Following Observation 9, as α_1 and α_2 are P-only, and since $\mathcal{D}$ satisfies Condition 1 for P, $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$ satisfies Condition 1 for $P \setminus \{p\}$. Condition 2 trivially holds since a process is removed from P and a register is added to S.

Condition 3 is satisfied for all processes in $\Pi \setminus P$ and configurations in $(\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2)$ as α_1 and α_2 are P-only, and since $\mathcal{D}$ satisfies Condition 3 for any process in $\Pi \setminus P$. As configurations in $\mathcal{D}$ are indistinguishable to p, p may only cover r_1 if $D \in \mathcal{D}\alpha_1$ and cover r_2 if $D \in \mathcal{D}\alpha_2$. But as given any $D \in \mathcal{D}$ we have $I(\{D\alpha_1, D\alpha_2\}, \Pi \setminus \{p\})$ (as $I(\mathcal{D}, P)$ following Observation 9), Condition 3 is also satisfied for p.

Since Consition 1, 2 and 3 are satisfied, we can thus apply Lemma 11 to obtain that $Confused(P \setminus \{p\}, S \cup \{r_2\}, (\mathcal{D}\alpha_1) \cup (\mathcal{D}\alpha_2))$. Indeed, a partition of two non-empty subsets of $S \cup \{r_2\}$ can be reduced to a partition of two non-empty subsets of S unless it is the partition S and $\{r_2\}$. But p may cover either $r_1 \in S$ or r_2. ◀

4.3 The lower bound

To establish our lower bound, we show that there is a set of reachable configuration $\mathcal{D}$ in which there is a process confused on all n registers. Intuitively, we proceed by induction on the number of "confusing" registers. For the base case, we show that the initial configuration can lead to a confusion of all but one process on *two registers*:

▶ **Lemma 14.** $\exists \mathcal{D} \in \mathit{Reach}(), \exists p \in \Pi, \exists S \subseteq \mathcal{R} : \mathit{Confused}(\Pi \setminus \{p\}, S, \mathcal{D})$.

Proof. Consider any two processes p_1 and p_2. Since the algorithm is comparison-based, the first write the two processes perform in a solo execution is on the same register, let us call it r. Let p_1 execute solo until it is about to write to r and then do the same with p_2, let C_1 be the resulting configuration. Consider the execution α from C_1 in which p_1 executes until it is poised to write to a register $r' \neq r$ and then p_2 executes its pending write on r. This execution is valid as p_1 must eventually write to an uncovered register.

As p_1 is hidden in α, $C_1\alpha$ and $C_1\alpha|_{\{p_2\}}$ are indistinguishable for all processes except p_1 (Observation 10). In $C_1\alpha$, p_1 is poised to write on r', but in $C_1\alpha|_{\{p_2\}}$, p_1 is poised to write on r, thus we have $\mathit{Confused}(\Pi \setminus \{p_1\}, \{r, r'\}, \{C_1\alpha, C_1\alpha|_{\{p_2\}}\})$. ◀

We now prove our inductive step. Given a set of configurations in which a set P of processes is confused on a set S of registers, we can obtain a set of configurations in which a set P' of processes are confused on a set S' of strictly more than $|S|$ registers:

▶ **Lemma 15.** $\exists \mathcal{D} \subseteq \mathit{Reach}(), P \subsetneq \Pi, S \subsetneq \mathcal{R} : \mathit{Confused}(P, S, \mathcal{D})$
$\implies \exists \mathcal{D}' \subseteq \mathit{Reach}(), P' \subseteq \Pi, S' \subseteq \mathcal{R}, S \subsetneq S' : \mathit{Confused}(P', S', \mathcal{D}')$.

Proof. Given $\mathit{Confused}(P, S, \mathcal{D})$, consider $C \in \mathcal{D}$ such that exactly $|S| - 1$ registers in S are covered by processes in $\Pi \setminus P$. Then we can reach a configuration in which all registers not in S are covered by processes in P. Indeed, when executed solo starting from C, a process must eventually write to a register that is not covered in C. Thus, it must eventually write either to a register in $\mathcal{R} \setminus S$ or to the uncovered register in S. Recall that, as $|S| + |P| = n + 1$, we have $|\mathcal{R} \setminus S| = |P| - 1$. Thus, by concatenating solo executions of processes in P until they are poised to write to uncovered registers, we reach a configuration $C\alpha$ in which *all* registers are covered, and let p be the process in P covering a register in S. Note that, as α is P-only, we have $\mathit{Confused}(P, S, \mathcal{D}\alpha)$. Thus:

$$\mathit{Cover}(\mathcal{R} \setminus S, P \setminus \{p\}, C\alpha) \wedge \mathit{Confused}(P, S, \mathcal{D}\alpha\}).$$

Now from this set of configurations, we are going to build a new one in which P is confused on *two* distinct sets of registers. By Lemma 12, there exist $p_c \in \Pi \setminus P$, $r \in S$ and $\mathcal{D}' \subseteq \mathcal{D}$ such that $\mathit{Confused}(P \cup \{p_c\}, S \setminus \{r\}, \mathcal{D}')$. Note that in $\mathcal{D}'$, the state of p_c can be chosen to be the state from any configuration from $\mathcal{D}$, and as we have $\mathit{Confused}(P, S, \mathcal{D})$ there must exists a configuration $C' \in \mathcal{D}$ in which p_c covers a register $r_c \in S \setminus \{r\}$. Let us select $\mathcal{D}'$ such that $C' \in \mathcal{D}'$.

If p is executed solo from $C'\alpha$, it must write infinitely often to *all* registers in S to ensure that it writes to an uncovered register. In a $\{p, p_c\}$-only execution from $C'\alpha$, p_c can hidden for arbitrarly many steps as long as p_c does not write to a register outside of S. But, as the algorithm satisfies 2-obstruction-freedom, p_c *must* eventually write to a register outside of S in such an execution. Consider the $\{p, p_c\}$-only execution β from $C'\alpha$ in which p_c is hidden and such that p_c execute until it is poised to write to some register $r' \in \mathcal{R} \setminus S$. Thus, we get two configurations $C'\alpha\beta$ and $C'\alpha\beta|_P$,indistinguishable to all processes but p_c, in which p_c covers, respectively, $r' \in \mathcal{R} \setminus S$ and $r_c \in S$. Thus, the conditions of Lemma 13 hold for $\mathcal{D}'$,

p_c, $\alpha\beta$ and $\alpha\beta|_{\{p\}}$ and so we obtain $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, (\mathcal{D}'\alpha\beta) \cup (\mathcal{D}'\alpha\beta|_{\{p\}}))$. As β is $\{p, p_c\}$-only and p_c is hidden in it, we have:

$$\mathit{Cover}(\mathcal{R} \setminus S, P \setminus \{p\}, C\alpha\beta) \wedge \mathit{Confused}(P, S, \mathcal{D}\alpha\beta|_{\{p\}}) \wedge$$

$$\mathit{Confused}(P, S \cup \{r'\} \setminus \{r\}, (\mathcal{D}'\alpha\beta) \cup (\mathcal{D}'\alpha\beta|_{\{p\}})).$$

Moreover, all configurations in the formula above are indistinguishable to processes in P, because $\alpha\beta$ is $P \cup \{p_c\}$-only and p_c is hidden in it (Observation 10).

Let $p' \in P$ be the process that covers r' in $C\alpha\beta$. According to p or p', every proper subset of S or $S \cup \{r'\} \setminus \{r\}$ may be covered at the same time by processes in $\Pi \setminus (P \cup \{p, p\})$, and all other registers are covered by processes in $P \setminus \{p, p'\}$. Thus, from $C\alpha\beta$, to complete a Write, p or p' must write to *all* registers in one of the sets S, $S \cup \{r'\} \setminus \{r\}$ or $\{r, r'\}$.

Consider any $\{p, p'\}$-only extension of $C\alpha\beta$. If one of $\{p, p'\}$ covers a register in $S \setminus \{r\}$, r or r', then the other process, in any solo extension, must write respectively to all registers in $\{r, r'\}$, S or $(S \cup \{r'\}) \setminus \{r\}$. In particular, since p' covers r' in $C\alpha\beta$, p running solo from $C\alpha\beta$ must eventually cover a register in $S \setminus \{r\}$ (as $|S| > 1$, since $|P| < n$ and $|P| + |S| = n + 1$). Then p' executing solo afterwards must write to r and r'. Let us stop p' when it covers a register $r'' \neq r$ for the last time before writing to r. Let γ be the resulting execution, and $E = C\alpha\beta\gamma$ be the resulting configuration. Let $\mathcal{E}$ and $\mathcal{E}'$ denote the sets of configurations indistinguishable from E to P defined as $\mathcal{D}\alpha\beta|_{\{p\}}\gamma$ and $(\mathcal{D}'\alpha\beta\gamma) \cup (\mathcal{D}'\alpha\beta|_{\{p\}}\gamma)$ respectively.

Now the following two cases are possible:

1. $r'' \notin S \cup \{r'\}$: In this case, we let p continue until it is poised to write on r, and then, we let the process in $P \setminus \{p, p'\}$ which covers r'' to proceed to its write. Let δ be this $\{p, p'\}$-only execution from E. As p' covers $r \in S$ in $E\delta$ and $r'' \in \mathcal{R} \setminus S$ in $E\delta|_{P \setminus \{p'\}}$, as $I(\{E\delta, E\delta|_{P \setminus \{p'\}}\}, \Pi \setminus \{p'\})$, and as $\mathit{Confused}(P, S, \mathcal{E})$, we can apply Lemma 13 and obtain $\mathit{Confused}(P \setminus \{p'\}, S \cup \{r''\}, (\mathcal{E}\delta) \cup (\mathcal{E}\delta|_{P \setminus \{p'\}}))$.
2. $r'' \in S \cup \{r'\}$, and so $r'' \in (S \cup \{r'\}) \setminus \{r\}$: Then we have the following sub-cases:
   - Some step performed by p in its solo execution from E makes p' to choose a register other than r to perform its next write in its solo extension. Clearly, this step of p is a write. From the configuration in which p is poised to execute this "critical" write, let p' run solo until it is poised to write to r and then let p complete its pending write. Let $E\delta$ be the resulting configuration.

     Now consider the execution in which p completes its "critical" write, then p' runs solo until it covers a register $r''' \neq r$. Let $E\delta'$ be the resulting configuration. Note that the states of the memory in $E\delta$ and $E\delta'$ are identical. Thus, $I(\{E\delta, E\delta'\}, \Pi \setminus \{p'\})$. Note that δ and in δ' are $\{p, p'\}$-only executions, and that p' covers r in $E\delta$ and r''' in $E\delta'$.
     a. If $r''' \in S$, as we have $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, \mathcal{E}')$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'\}), (\mathcal{E}'\delta) \cup (\mathcal{E}'\delta'))$.
     b. If $r''' \in \mathcal{R} \setminus S$, as we have $\mathit{Confused}(P, S, \mathcal{E})$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'''\}), (\mathcal{E}\delta) \cup (\mathcal{E}\delta'))$.
   - Otherwise, no write of p is "critical", and we let it run from E until it covers r (recall that, as p' covers $r'' \in (S \cup \{r'\}) \setminus \{r\}$, p must eventually write to all registers in S or $\{r, r'\}$ and, thus, to r). Let then p' run until it covers r and let δ be the resulting execution. From $E\delta$, let p' run until it becomes poised to write to a register $r''' \neq r$ for the first time, and then let p perform its pending write on r. Let λ be this extension. Note that as p' is hidden in λ, we have $I(\{E\delta\lambda, E\delta\lambda|_{\{p'\}}\}, \Pi \setminus \{p\})$. Also, δ and λ are $\{p, p'\}$-only executions such that p' covers r in $E\delta\lambda|_{\{p\}}$ and covers r''' in $E\delta\lambda$.
     a. If $r''' \in S$, as we have $\mathit{Confused}(P, (S \cup \{r'\}) \setminus \{r\}, \mathcal{E}')$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'\}), (\mathcal{E}'\delta\lambda) \cup (\mathcal{E}'\delta\lambda|_{\{p\}}))$.

b. If $r''' \in \mathcal{R} \setminus S$, as we have $\mathit{Confused}(P, S, \mathcal{E})$, applying Lemma 13, we obtain $\mathit{Confused}(P \setminus \{p'\}, (S \cup \{r'''\}), (\mathcal{E}\delta\lambda) \cup (\mathcal{E}\delta\lambda|_{\{p\}}))$. ◀

Our lower bound directly follows from Lemmata 14 and 15:

▶ **Theorem 16.** *Any n-process comparison-based 2-obstruction-free SWMR memory implementation requires $n + 1$ MWMR registers.*

Proof. By contradiction, suppose that an n-register algorithm exists. We show, by induction, that there is a reachable configuration in which a process is confused on *all* registers. Lemma 14 shows that there exists a reachable configuration in which $n - 1$ processes are confused on two registers. We can therefore apply Lemma 15 and obtain a configuration with a confusion with strictly more registers. By induction, there exist then a set of configurations $\mathcal{D}$ and $p \in \Pi$ such that $\mathit{Confused}(\{p\}, \mathcal{R}, \mathcal{D})$.

Thus, any strict subset of $\mathcal{R}$ is covered by the remaining $n - 1$ processes in some configuration in the (indistinguishable for p) set of configurations $\mathcal{D}$. But a process p may complete a Write operation if and only its write value is present in a register which is not covered (by a process not aware of the value) in any of the configurations indistinguishable to p. Therefore, in an infinite solo execution of p, p must write infinitely often to *all* registers. But then any arbitrarily long execution by any other process can be hidden by incorporating sufficiently many steps of p , violating 2-obstruction-freedom—a contradiction. ◀

5 Concluding remarks

This paper shows that the optimal space complexity of SWMR implementations depends on the desired progress condition: lock-free algorithms trivially require n registers, while 2-obstruction-free ones (and, thus, also 2-lock-free ones) require $n + 1$ registers. We also extend the upper bound to k-lock-freedom, for all $k = 1, \ldots, n$, by presenting a k-lock-free SWMR implementation using $n + k - 1$ registers. A natural conjecture is that the algorithm is optimal, i.e., no such algorithm exists for $n + k - 2$ registers for all $k = 1, \ldots, n$. Since for $k = 1$, 2 and n, k-obstruction-freedom and k-lock-freedom impose the same space complexity, it also appears natural to expect that this is also true for all $k = 1, \ldots, n$.

An interesting corollary to our results is that to implement a 2-obstruction-free SWMR memory we need strictly more space than to implement a 1-lock-free one. But the two properties are, in general, incomparable: a 2-solo run in which only one process makes progress satisfies 1-lock-freedom, but not 2-obstruction-freedom, and a run in which 3 or more processes are correct but no progress is made satisfies 2-obstruction-freedom, but not 1-lock-freedom. The relative costs of incomparable progress properties, e.g., in the (ℓ, k)-freedom spectrum [5], are yet to be understood.

An SWMR memory can be viewed as a *stable-set* abstraction with a conventional *put/get* interface: every participating process can put values to the set and get the set's content, and every get operation returns the values previously put. For the stable-set abstraction, we can extend our results to the *anonymous* setting, where processes are not provided with unique identifiers. Indeed, we claim that the same algorithm may apply to the stable-set abstraction for anonymous systems when the number of participating processes n is known. But the question of whether an *adaptive* solution exists (expressed differently, a solution that does not assume any upper bound on the number of participating processes) for anonymous systems remains open.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990. doi:10.1145/79147.79158.

3 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. In *19th International Conference on Principles of Distributed Systems*, OPODIS '15, pages 18:1–18:17, 2015.

4 James E Burns and Nancy A Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

5 Victor Bushkov and Rachid Guerraoui. Safety-liveness exclusion in distributed computing. In *34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 227–236, 2015.

6 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *International Symposium on Distributed Computing*, DISC '13, pages 269–283, 2013.

7 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Black art: Obstruction-free k-set agreement with |mwmr registers| < |proccesses|. In *1st International Conference on Networked Systems*, NETYS '13, pages 28–41, 2013.

8 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133, 2015.

9 Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the space complexity of set agreement? In *34th ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 271–280, 2015.

10 Panagiota Fatourou, Faith Ellen Fich, and Eric Ruppert. Time-space tradeoffs for implementations of snapshots. In *38th ACM Symposium on Theory of Computing*, STOC '06, pages 169–178, 2006.

11 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, 2003.

12 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for non-blocking implementations (preliminary version). In *15th ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 257–266, 1996.

13 Leslie Lamport. On interprocess communication; part I and II. *Distributed Computing*, 1(2):77–101, 1986.

14 F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

15 Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *23rd International Conference on Distributed Computing*, DISC'09, pages 157–171, 2009.

16 Nayuta Yanagisawa. Wait-free solvability of colorless tasks in anonymous shared-memory model. In *18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '06, pages 415–429, 2016.

17 Leqi Zhu. A tight space bound for consensus. In *48th ACM Symposium on Theory of Computing*, STOC '16, pages 345–350, 2016.

A Proofs for Algorithm 1 (k-lock-free SWMR implementation)

▶ **Lemma 2.** *Let, at some point of a run of the algorithm, value (v, id, c) be present in some register r and such that no process is poised to execute an update on r (i.e., no process is between taking the snapshot of MEM (line 9) and the update of r (line 12)), then at all subsequent times $(v, id, c) \in r$, i.e., the value is present in the set of values stored in r.*

Proof. Suppose that at time τ, a register R contains (v, id, c) and no process is poised to execute an update on R. Suppose, by contradiction, that R does not contain it at some time $\tau' > \tau$. Let τ_{min}, $\tau_{min} > \tau$, be the smallest time such that (v, id, c) is not in R. Therefore, a write must have been performed on R, by some process q, at time τ_{min} with a view which does not contain (v, id, c). Such a write can only be performed at line 12, with a view including the last snapshot of MEM performed by q at line 9. Process q must have performed this snapshot operation on R at some $\tau_R < \tau$ as (v, id, c) is present in R between times τ and τ_{min} and as $\tau_R < \tau_{min}$. Thus q is poised to write on R at time τ — a contradiction. ◀

▶ **Lemma 3.** *If process p returns from a Write operation $(v, id(p), c)$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(p), c)$.*

Proof. Before returning from its Write operation, p takes a snapshot of MEM at some time τ_S, $\tau_S < \tau$ (line 9), which returns a view of the memory in which at least n registers contain the triplet $(v, id(p), c)$. As p is taking a snapshot at line 9 at time τ_S, at most $n-1$ processes can be poised to perform an update on some register at time τ_S. As a process can be poised to perform an update on at most one register at a time, there can be at most $n-1$ distinct registers covered at time τ_S. Therefore, at time τ_S, there is at least one uncovered register containing $(v, id(p), c)$, let us call it r. By Lemma 2, $(v, id(p), c)$ will be present in r at any time $\tau' > \tau_S$, and thus at any time $\tau' > \tau$ as $\tau > \tau_S$. ◀

▶ **Lemma 5.** *Write operations in Algorithm 1 satisfy 1-lock-freedom.*

Proof. Suppose, by way of contradiction, that Write operations do not satisfy 1-lock-freedom. Then, eventually, all n first registers are infinitely often updated only by correct processes unsuccessfully trying to complete a Write operation. Thus, eventually each of the n first registers contain the value from one of these incomplete Write operations. As there are at most $n-1$ covered registers when a snapshot is taken, one of these value is eventually permanently present in some register (Lemma 2). This value is then eventually contained in the local view of every correct process, and thus, will eventually be present in every update of all the n first registers. The process with this Write value must therefore eventually pass the test on line 15 and, thus, complete its Write operation — a contradiction. ◀

▶ **Lemma 6.** *If a process q performing infinitely many operations sees $(v, id(p), c)$, and if p is correct, then p eventually completes its c^{th} Write operation.*

Proof. By Lemma 3, if process q returns from a Write operation with value $(v, id(q), c')$ at time τ, then for any time $\tau' \geq \tau$ there is a register containing $(v, id(q), c')$. But note that $(v, id(q), c')$ is written to a register only associated with q's local view. Thus, as q completes an infinite number of Write operations, each local view of q will eventually be forever present in some register. It will then eventually be observed in every snapshot taken by correct processes, and, therefore, included in their local view. This implies that it will eventually be present in every register written infinitely often, in particular in the first n registers. Process p will then eventually pass the test at line 15 and complete its corresponding c^{th} Write operation. ◀

The Teleportation Design Pattern for Hardware Transactional Memory

Nachshon Cohen[1], Maurice Herlihy[2], Erez Petrank[3], and Elias Wald[4]

1 **Computer Science Dept., EPFL, Lausanne, Switzerland**
nachshonc@gmail.com
2 **Computer Science Dept., Brown University, Providence, RI, USA**
mph@cs.brown.edu
3 **Computer Science Dept., Technion, Haifa, Israel**
erez@cs.technion.ac.il
4 **Computer Science Dept., Brown University, Providence, RI, USA**
elias_wald@brown.edu

Abstract

We identify a design pattern for concurrent data structures, called *teleportation*, that uses best-effort hardware transactional memory to speed up certain kinds of legacy concurrent data structures. Teleportation unifies and explains several existing data structure designs, and it serves as the basis for novel approaches to reducing the memory traffic associated with fine-grained locking, and with hazard pointer management for memory reclamation.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.1.3. Concurrent Programming, E.1 Data Structures

Keywords and phrases Hardware transactional memory, concurrent data structures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.10

1 Introduction

The emerging widespread availability of *hardware transactional memory* (HTM) [18] on commodity processors such as Intel's x86 [20] and IBM's Power [7] architectures provides opportunities to rethink the design and implementation of highly-concurrent data structures.

This paper identifies the *teleportation* design pattern, which uses best-effort hardware transactional memory to reduce the memory costs of certain kinds of concurrent data structures. Retrospectively, the teleportation pattern provides a common framework for describing and explaining several heretofore unrelated techniques from the literature. Here, we describe two novel applications of this design pattern: to reduce the memory overheads associated with fine-grained locking, and with hazard pointer management for memory reclamation.

In highly-concurrent data structures, a high-level operation (such as adding a value to a hash set) is implemented as a sequence of atomic state transitions, where each transition is identified with a *memory operation*. Memory operations include loads, (non sequentially-consistent) stores, memory barriers, and atomic operations such as compare-and-swap (CAS). Operations of concurrent threads can be interleaved.

This model encompasses both lock-based and lock-free data structures. For example, a thread applying an operation to a lock-based data structure might spin until the lock is free (repeated loads), acquire that lock (CAS, with implicit barrier), modify the data

structure (loads and stores), and finally release the lock (store and memory barrier). A thread traversing a lock-free data structure might read fields (loads, perhaps barriers) and perform one or more CAS operations.

Sometimes, a thread's sequence of memory operations has the property that if the thread were to execute that sequence uninterruptedly, that is, without interleaving steps with other threads, then some of the memory operations could be *elided* (omitted), because later memory operations "undo" the effects of earlier ones. For example, *Lock elision* [25, 26] is a well-known technique that exploits the observation that a matching lock acquisition and release "cancel" one another, leaving the lock's memory state unchanged at the end of the critical section. If a thread executes a critical section as a best-effort hardware transaction, then it can reduce memory traffic by eliding the lock acquisition and release.

The *teleportation pattern* retrofits an existing data structure with additional speculative code paths chosen to elide memory operations. When a thread reaches a state where a speculative alternative exists, it executes that path as a best-effort hardware transaction. If that transaction fails for any reason, its effects are automatically discarded,

Within the transaction, the thread appears to execute as if it were running uninterruptedly, with no interleaved steps by other threads. From the other threads' viewpoints, the speculating thread appears to transition instantaneously from the path's initial state to its final state, hence the name "teleportation".

As discussed in the related work section, examples of mutually-canceling memory operations include lock acquisition and release [25, 26], short-lived reference counters with matching increments and decrements [11], and sequences of memory barriers that can be coalesced into a single barrier [2].

This paper makes the following contributions. First, we believe that identifying the teleportation design pattern itself is a useful addition to the lexicon, as it explains and unifies several earlier techniques from the literature, and it provides a way to organize and explain the new techniques described here. To be useful, a design pattern should help unify and explain existing designs (*e.g.*,"both lock elision and telescoping are instances of the teleportation pattern"), and it should suggest specific approaches to new problems (*e.g.*, "can we use the teleportation pattern to improve hazard pointers?"). It should be general enough to apply to disparate situations, but not so general as to be meaningless.

Next, we show how the teleportation pattern can be applied to retrofit speculative fast paths to well-known fine-grained locking algorithms such as *lock-coupling* [4] (or hand-over-hand locking). Teleporting from one lock-holding state to another avoids the memory traffic produced by locking and unlocking intermediate nodes.

Finally, we show how the teleportation pattern can be applied to retrofit speculative fast paths to hazard pointer-based memory management schemes [24]. Teleporting from one hazard pointer-publishing state to another, avoids the memory barriers needed to publish intermediate nodes' hazard pointers.

Experimental results on a 8-core Broadwell show that teleportation can substantially boost the performance of fine-grained locking and lock-free hazard pointer management. In some cases, the hazard pointer scheme with teleportation outperforms the same structure with a faster but less robust epoch-based reclamation scheme.

2 List-Based Sets

Pointer-and-node data structures come in many forms: skiplists, hashmaps, trees, and others. Here, for simplicity, we consider *list-based sets*, using data structures adapted from the Herlihy and Shavit textbook [19]. There are three methods: $\mathsf{add}(x)$ adds x to the set, returning *true*

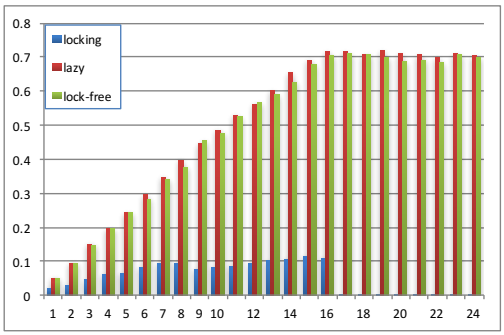


Figure 1 Throughput in Million Operations Per Second (MOPS) for list implementation without memory management.

if and only if x was not already there, remove(x) removes x from the set, returning *true* if and only if x was there, and contains(x) returns *true* if and only if the set contains x. (We will discuss trees and skiplists later.)

A set is implemented as a linked list of nodes, where each node holds an item (an integer), a reference to the next node in the list, and possibly other data such as locks or flags. In addition to regular nodes that hold items in the set, we use two *sentinel* nodes, called LSentinel and RSentinel, as the first and last list elements. Sentinel nodes are never added, removed, or searched for, and their items are the minimum and maximum integer values.

We consider three distinct list implementations: the *lock-based* list, the *lazy* list, and the *lock-free* list. These three list algorithms are discussed in detail elsewhere [19].

Our *lock-based* list implementation employs *fine-grained locking* and is based on *lock coupling* [4], sometimes called *hand-over-hand locking*. Each list node has its own lock. A thread traversing the data structure first locks a node, then locks the node's successor, then releases the predecessor's lock. While fine-grained locking permits more concurrency than using a monolithic lock, it causes threads to acquire and release many locks, and restricts parallelism because threads cannot overtake one another.

By contrast, in the so-called *lazy list*, each list node has a Boolean marked field indicating whether that node's value is really in the set. Threads navigate through the list without acquiring locks. The contains() method consists of a single wait-free traversal: if it does not find a node, or finds it marked, then that item is not in the set. To add an element to the list, add() traverses the list, locks the target's predecessor, and inserts the node. The remove() method is lazy, taking two steps: first, mark the target node, *logically* removing it, and second, redirect its predecessor's next field, *physically* removing it.

Finally, the *lock-free* list replaces the lazy list's lock acquisitions with atomic compare-and-swap operations.

To establish a baseline for understanding the costs of synchronization and memory management, we ran a simple synthetic benchmark comparing the lock-based, lazy, and lock-free list implementations. We used an Intel Broadwell processor (Core D-1540) with RTM and HLE enabled, running at 2 GHz. The machine has a total of 8GB of RAM shared across eight cores, each having a 32 KB L1 cache. Hyper-threading was enabled, yielding a total of sixteen hardware threads. Threads were not pinned to cores.

When there are 8 or fewer threads, each thread has its own core. Between 9 and 16 threads, hyperthreading ensures that each thread has its own hardware context, but threads share caches. Beyond 16 threads, there are more threads than hardware contexts.

Each thread allocates nodes by calling the **new** operator, but for now nodes are not reused after being removed from the list. We use a simple "test-and-test-and-set" spin lock, where a

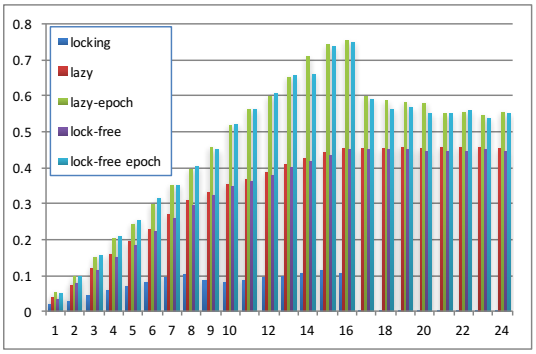


Figure 2 Throughput for list implementations with memory management.

thread spins reading the lock until it appears to be free, and then applies a compare-and-swap to try to acquire the lock. If that compare-and-swap fails, the thread temporarily releases the processor. To avoid false sharing, locks are padded so distinct locks occupy distinct cache lines.

The list implementations were compared using a simple synthetic benchmark based on the following parameters. List values range from zero to 10,000, and the list is initialized to hold approximately half of those values. All measurements are executed for one second. The number of threads varies from 1 to 24. Each time a thread calls an operation, it chooses contains() with a probability that varies from 0 to 100, and otherwise calls add() or remove() with equal probability.

All the experimental results reported here have the following structure. We ran experiments with 100%, 90%, 80%, 50%, and 0% contains() calls. Because the cost of list operations is dominated by navigation, all these mixtures produced similar curves. The graphs shown here reflect a mixture of 80% contains(), 10% add(), and 10% remove() calls. Each execution time shown in a graph represents an average over 10 executions.

Figure 1 compares the performance of the three list implementations on the benchmark *with no memory management*. This graph is a baseline allowing us to separate the costs of navigation (shown here) from the costs of memory management (shown later).

The lock-based list performs substantially worse than the other two because it acquires and releases a lock each time it traverses a node, while the others navigate without locks, acquiring locks or executing CAS operations only at the very end. Beyond 16 threads, there are more threads than hardware contexts, and fine-grained locking becomes extremely slow because some thread is always suspended while holding a lock, and while that thread is suspended, no active thread can overtake it.

In the next section, however, we will see that Figure 1, while informative, can be misleading, because none of these implementations performs any memory management.

3 Memory Management

We now add memory management to the three list implementations considered in the previous section. To provide a common basis for comparison, we use the following simple structure. Each thread keeps a thread-local pool of *free* nodes available for immediate reuse, and a thread-local pool of *retired* nodes that have been removed from the list, but which may not be ready to be reinitialized and reused. Both pools are implemented as lists, but because they are thread-local, they do not require synchronization.

When a thread needs to allocate a node to add to the list, it first tries to take a node from its free pool. If the free pool is empty, it inspects the retired pool to determine whether there are any nodes that can be moved to the free pool. If not, it calls the C++ **new** operator to allocate a new node.

For the lock-coupling list, memory management is simple: as soon as a node is removed from the list, it can be placed directly in the free pool, bypassing the retired pool.

For the lazy and lock-free lists, however, a node cannot be reused immediately after it is retired because another thread, performing a lock-free traversal, may be about to read from that node, presenting a so-called *read hazard*.

Hazard pointers [24] provide a non-blocking way to address this problem. When a thread reads the address of the node it is about to traverse, it *publishes* that address, called a *hazard pointer*, in a shared array. It then executes a *memory barrier*, ensuring that its hazard pointer becomes immediately visible to the other threads. It rereads the address, and proceeds if the address is unchanged, As long as the hazard pointer is stored in the shared array, a thread that has retired that node will not reinitialize and reuse it. The memory barrier is the most expensive step of this protocol.

For the lazy and lock-free lists with hazard pointers, each thread requires two hazard pointers, one for the current node, and one for the previous node. To avoid false sharing, shared hazard pointers are padded so that hazard pointers for different threads reside on different cache lines.

For purposes of comparison, we also test an *epoch-based* reclamation scheme [12]. The threads share a global *epoch count* to determine when no hazardous references exist to a retired node in a limbo list. Each time a thread starts an operation that might create a hazardous reference, it observes the current epoch. If all processes have reached the current epoch, the global epoch can be advanced, and all nodes that were retired two epochs ago can safely be recycled. A thread starting such an operation blocks only if its own free list is empty, but there are retired nodes to be reclaimed when trailing threads catch up to the current epoch. Each thread needs to keep track of retired pools the last three epochs only. Careful signaling ensures that quiescent threads do not inhibit reclamation, but, as noted, an active thread that encounters a delay inside an operation can prevent every other thread from reclaiming memory.

Figure 2 shows the same benchmark for the same list implementations, except now with memory management. The lazy and lock-free lists with hazard pointers have become significantly slower because of the need to execute a memory barrier each time a node is traversed, an operation almost as expensive as acquiring a lock. The lock-coupling implementation is still slower than all, but it is significantly simpler than the other methods that require non-trivial memory management code. Epoch-based reclamation does not significantly slow down these data structures (but of course the "lock-free" list is no longer lock-free).

4 Teleportation

Teleportation for lock-based data structures works as follows: a thread holding the lock for a node launches a hardware transaction that traverses multiple successor nodes, acquires the lock only for the last node reached, and then releases the lock on the starting node before it commits. (For lock-free and lazy algorithms, the same observations hold, except replacing lock acquisition and release with hazard pointer publication.)

Compared to simple fine-grained locking, teleportation substantially reduces the number of lock acquisitions (or hazard pointer memory barriers) needed for traversals. Moreover, if

a hardware transaction that attempts to teleport across too many nodes fails, the thread can adaptively try teleporting across fewer nodes, and in the limit, it reverts to conventional fine-grained locking manipulation.

4.1 Intel RTM

Although teleportation can be adapted to multiple HTM schemes, we focus here on Intel's *transactional synchronization extension* (TSX) [20], which supports hardware transactions through the *restricted transactional memory* (RTM) interface.

When a hardware transaction begins execution, the architectural state is checkpointed. Memory operations performed inside a transaction are tracked and buffered by the hardware and become visible to other threads only after the transaction successfully commits. The hardware detects memory access conflicts with other threads, both transactional and non-transactional. If a conflict is detected, one transaction *aborts*: its buffered updates are discarded and its state is restored.

In the RTM interface, a thread starts a hardware transaction by calling xbegin. If the call returns XBEGIN_STARTED, then the caller is executing inside a hardware transaction. Any other return value means that the caller's last transaction aborted. While inside a transaction, calling xend() attempts to commit the transaction. If the attempt succeeds, control continues immediately after the xend() call in the normal way. If the attempt fails, then control jumps to the return from the xbegin call, this time returning a code other than XBEGIN_STARTED. While inside a transaction, calling xabort() aborts the transaction. A thread can test whether it is in a transaction by calling xtest().

4.2 Teleportation Algorithm

The heart of the teleportation algorithm is the teleport() method shown in Listing 1. (For brevity, we focus on lock-based teleportation.) This method takes three arguments: a starting node, the target value being sought, and a structure holding thread-local variables. The arguments must satisfy the following preconditions: the starting node must be locked by the caller, and the starting node's successor's value must be less than the target value. When the method returns, the starting node is unlocked, the result node whose address is returned is locked, and the result node's value is greater than the starting node's value, but less than the target value.

The thread-local teleportLimit variable is the maximum distance the method will try to teleport the lock. (In our experiments, we chose an initial value of 256, but the results were insensitive to this choice.) The method initializes pred to the start node, and curr to the start node's successor (Lines 4–5). It then enters a loop. At the start of each iteration, it starts a hardware transaction (Line 7), which advances pred and curr while curr−>value is less than the target value (Lines 10–15), stopping early if more than teleportLimit nodes are traversed (Line 14). On leaving the loop, the starting node is unlocked and pred is locked, teleporting the lock from one to the other (Lines 16–17). If the transaction commits (Line 18), then the limit for the next transaction is increased by 1 (Line 19), and the method returns pred.

If the commit fails, or if the transaction explicitly aborts, then the teleportation distance is halved, and the outer loop is resumed (Line 21). However, we do not let the distance fall below a threshold (we chose 20[1]), because short teleportation distances harm performance, especially in the tree algorithm. When too many failures occur (more than 10), the algorithm reverts to standard fine-grained locking (Lines 28–30).

[1] The exact value of constants used in our teleportation code had little effect on overall performance.

Listing 1 Lock-based List: the teleport() method.

```
Node* teleport(Node* start,
               T v,
               ThreadLocal* threadLocal) {
  Node* pred = start;
  Node* curr = start->next;;
  int retries = 10;
  while (--retries) {
    int distance = 0;
    if (xbegin() == _XBEGIN_STARTED) {
      while (curr->value < v) {
        pred = curr;
        curr = curr->next;
        if (distance++>threadLocal->teleportLimit)
          break;
      }
      pred->lock();
      start->unlock();
      _xend();
      threadLocal->teleportLimit++;
      return pred;
    } else {
      threadLocal->teleportLimit =
            threadLocal->teleportLimit/2
      if(threadLocal->teleportLimit<THRESHOLD)
        threadLocal->teleportLimit=THRESHOLD;
    }
  }
  curr->lock();
  start->unlock();
  return curr;
};
```

As noted, this method follows an *additive increase, multiplicative decrease* (AIMD) strategy: the teleportation limit is incremented by 1 upon each successful commit, and halved upon each abort. We experimented with several other adaptive policies without noticing a dramatic difference. Overall, AIMD proved slightly better than the others.

Because atomic operations such as compare-and-swap are typically not permitted inside hardware transactions, The spinlock's lock() method calls xtest() to check whether it is in a transaction. If so, and if the lock is taken, then the method explicitly aborts its transaction. If, instead, the lock is free, it simply writes the value LOCKED and returns. Otherwise, if the caller is not in a transaction, then it executes a simple "test-and-test-and-set" lock as before.

When teleport() traverses a node, it reads the node's value and next fields, but not the node's spinLock field. (These fields are cache-aligned to avoid false sharing.) This omission means that one teleporting thread can overtake another, which is impossible with conventional fine-grained locking.

Listing 2 Lock-based list: remove() with teleportation.

```
1  bool remove(T v, ThreadLocal* threadLocal) {
2    Node* pred = &LSentinel;
3    bool result = false;
4    int retryCount = 0;
5    while (true) {
6      if (pred->next->value > v) {
7        result = false;
8        break;
9      } else if (pred->next->value == v) {
10       Node *nodeToDelete = pred->next;
11       nodeToDelete->lock();
12       pred->next = nodeToDelete->next;
13       free(nodeToDelete);
14       result = true;
15       break;
16     } else {
17       pred = teleport(pred, v, threadLocal);
18     }
19   }
20   pred->unlock();
21   return result;
22 }
```

The teleport() methods for the lazy and lock-free lists are similar, omitting hazard pointer publication instead of locking.

4.3 How to teleport()

The lock-based remove() method, shown in Listing 2, sets pred to the local node (as discussed above) and enters a loop (Lines 5–19). If the pred node's successor's value is greater than the target value, then the value is not present in the list, so the method returns *false* (Line 6). If the pred node's successor's value is equal to the target value, then we lock the successor and unlink it from the list (Line 9). Finally, if the pred node's successor's value is less than the target value, we call teleport() to advance pred, and resume the loop (Line 16).

The tree-based set's range query method uses lock coupling. The range query method takes a pair (k_1, k_2) and returns the set of all items in the set between k_1 and k_2. Range queries are difficult for lists, but conceptually easy for unbalanced binary trees. Lock-coupling provides an easy snapshot view of the data structure, but it performs poorly because it acquires a lock for each traversed node, significantly reducing concurrency. Teleportation improves performance, while maintaining the simplicity and the functionality of the lock-coupling algorithm. The teleporting thread checks the traversed nodes' locks and aborts if any is locked. Since threads do not bypass locked nodes, locking a node freezes its subtree, allowing a snapshot of the key range.

The range query method starts by searching for the lowest-level node whose subtree contains all elements in the range. This search employs lock-coupling with teleportation. Once the lock of the subtree's root is acquired, the method gathered the nodes of the subtree

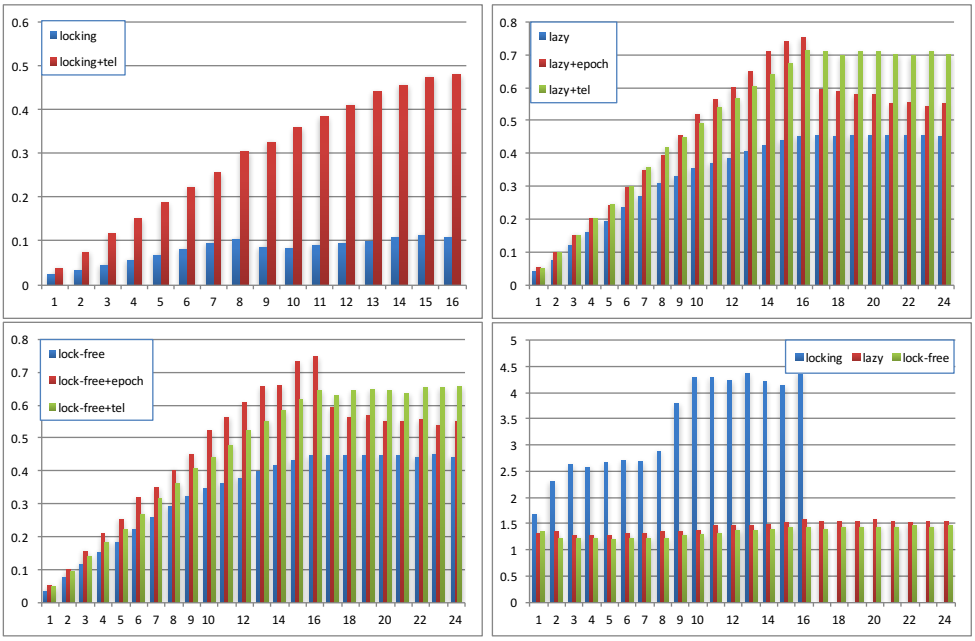


Figure 3 Effects of teleportation on various lists. Graphs (a)-(c) show throughput in million operations per second, higher is better. Graph (d) shows speedup, higher means teleportation is more effective.

that are in the desired range one by one, making sure that previous operations have completed and released the nodes' locks. Finally, the subtree's root lock is released.

The teleportation algorithm for the range queries is similar to the list teleportation algorithm in Listing 1, except that it checks the visited node's lock and aborts if the lock is already held. In addition, it goes to the right if the current key is smaller than the maximum, to the left if the current key is larger, and otherwise it acquires the target's lock and commits.

4.4 An Optimization

One standard disadvantage of using fine-grained locking with a tree is that the lock of the tree root becomes a bottleneck, serializing operations that could otherwise be executed in parallel. To avoid high contention on the root, we add a simple, yet effective, optimization. A precondition to the teleportation method is that the node it receives is locked. Our optimization lets each thread have an artificial local node pointing to the sentinel first node of the data structure, e.g., the root of the tree. The artificial local node is never reclaimed and is always locked by the thread using it. A thread starts a search from its local node, and teleports from there. This eliminates contention on the first node and has several disadvantages. If two operations teleport on separate routes from the tree root node, they will not conflict. But even for a list, since the teleportation length varies between threads, there are multiple target nodes for the first teleportation, and contention is reduced. To see that correctness is preserved, note that the node returned by the first teleportation is either the sentinel node or a node reachable from it.

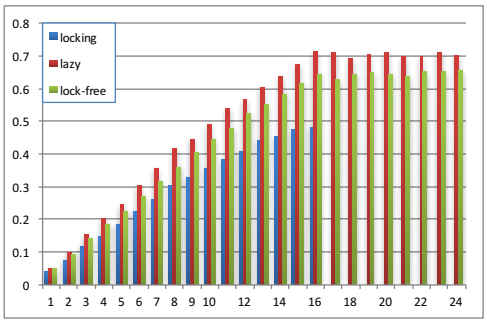


Figure 4 Throughput with teleportation.

4.5 Performance Measurements

Figure 3a shows how teleportation improves the performance of the lock-based list implementation. For a single thread, teleportation yields an approximate 1.5-fold speedup, increasing to a 4-fold speedup at 16 threads.[2]

Figure 3b compares lazy list with various memory management schemes. It compares the throughput for (1) the lazy list with hazard pointers, (2) the lazy list with epoch-based management, and (3) the lazy list with hazard pointers and teleportation. Teleportation speeds up lazy list with hazard pointers by 33% – 64%. Teleportation with hazard pointers is just as fast as epoch-based reclamation, while eliminating the epoch-based scheme's vulnerability to thread delay. It is slower by 6% for a single thread, faster by 0% – 5% for 2 – 8 threads, slower by 1% – 9% for 9 – 16 threads, and faster by 18% – 30% for 17 – 24 threads. We also experimented with adding teleportation to the epoch-based scheme, but we found that epoch-based performance is unaffected by teleportation because it does not perform memory barriers between most transitions.

Figure 3c compares lock-free list with various memory management schemes. It compares the throughput for (1) the lock-free list with hazard pointers, (2) the lock-free list with epoch-based management, and (3) the lock-free list with hazard pointers and teleportation. Teleportation speeds up the lock-free list with hazard pointers by 21% – 48%. When compared against epoch-based reclamation, teleportation is slightly slower at low thread counts (5% – 15%), and slightly faster (7% – 21%) above 16 threads.

Figure 3 provides a clear picture of the teleportation speedups in the above measurements. First, when applied to improve the performance of hand-over-hand locking; second, when applied to the hazard pointers with lazy list; and third, when applied to the hazard pointers scheme for the lock-free list.

Figure 4 compares the benchmark performances for the three list implementations, with memory management and with teleportation. The lazy implementation performs best for low thread counts - it is faster than the lock-free implementation by 7% – 13%. The locking implementation does not perform as well as lazy or lock-free lists, but it is easier to develop, maintain, and extend.

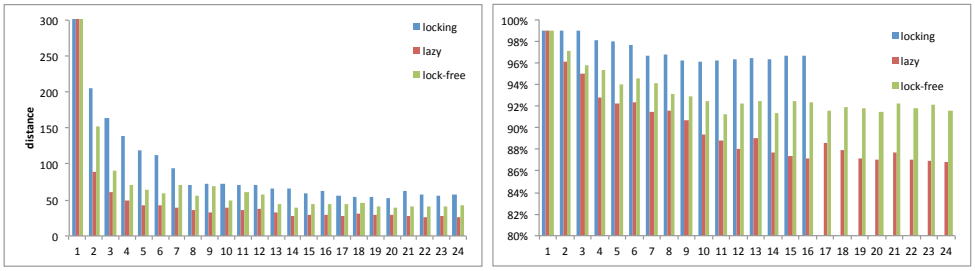


Figure 5 Average teleportation distance (L) and commit rates (R).

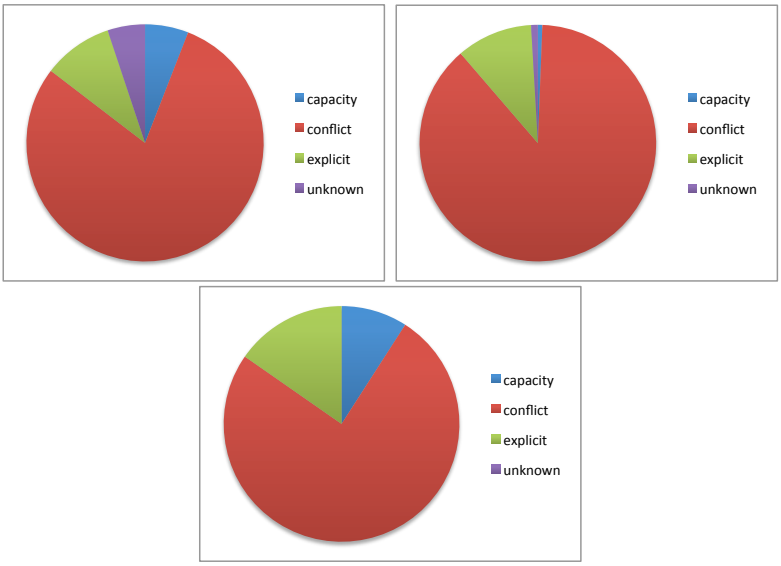


Figure 6 Why transactions abort: 4, 8, and 12 threads.

4.6 Transaction Measurements

Figure 5 presents the average distance (number of nodes) a lock or hazard pointer is teleported by each list algorithm, and Figure 5 shows the transaction commit rates. The lock-free implementation is able to teleport around 50 nodes and its commit rate is about 95%. The lazy implementation is able to teleport approximately 30 nodes at a transaction and the commit rate is around 88%. The locking implementation has the highest commit rate, around 96, and average teleportation distance around 70.

Although only small percentage of transactions abort, it is instructive to examine why. When a transaction aborts in RTM, it stores a condition code that indicates (more-or-less) why it aborted. In our benchmarks, we observed four kinds of abort codes.

- A *capacity* abort occurs if a transaction's data set overflows the L1 cache for any reason.
- A *conflict* abort occurs if one transaction writes to a location read or written by another transaction.
- An *explicit* abort occurs when the software calls xabort(). Here, this kind of abort means that a transaction tried to lock a lock already held by another thread.
- An *unknown* abort is any other kind of abort.

[2] Fine-grained locking becomes very slow beyond 16 threads, but fine-grained locking with teleportation works well beyond this limit because teleporting threads can overtake one another.

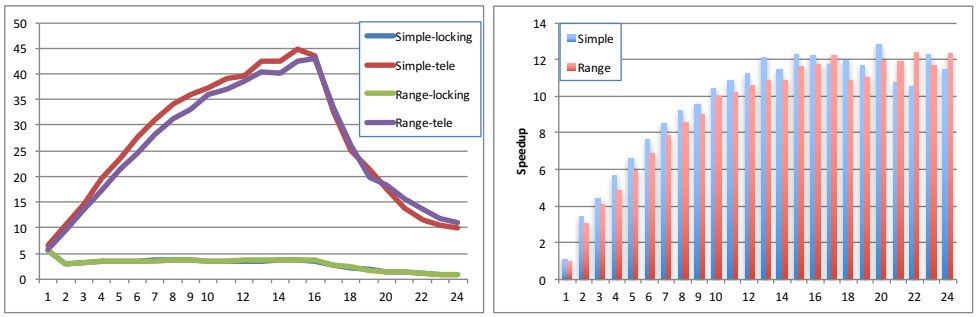


Figure 7 Throughput and speedup for binary tree with range queries.

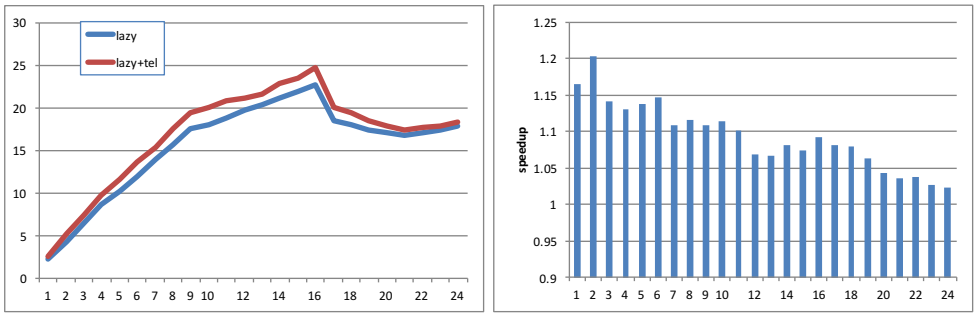


Figure 8 Throughput and speedup for skip list.

Figure 6 shows the distribution of abort codes for the lock-based list teleportation, for 4, 8, and 12 threads. Most aborts are conflict and explicit aborts, which is a good result, meaning that aborts are caused mostly by real conflicts and not by hardware limitations or inaccuracies.

In addition to the linked list, we also implemented a skiplist based on the lazy list with teleportation, and a binary tree with range queries based on lock-coupling. For both implementations we used a micro-benchmark with a typical workload of 10% insertions, 10% deletions, and 80% contain operations. For the binary tree we further added a range workload, where the contain operations are replaced by range queries of length 10. In Figure 7 we report execution times for the binary tree with range queries support, implemented with the lock-coupling paradigm. It can be seen that the lock-coupling algorithm offers no scalability. However, with teleportation, excellent scalability is obtained. In Figure 7 we present the speedup of teleportation over the standard lock-coupling algorithm. With a single thread there is almost no improvement, but for two threads and up the speedup is 3.1 – 12.

Finally, in Figure 8 and Figure 8 we consider the speedup and execution time of a lazy skip list with and without hazard pointers teleportation. The teleportation speeds up the lazy implementation by 2% – 20%.

5 Correctness

Although a formal model of teleportation is beyond the scope of this paper, we can sketch what such a treatment would look like. There are multiple *threads*, where each thread is a *state machine* that undergoes a sequence of state *transitions*. The threads communicate through a (passive) shared *memory*. Each thread transition is associated with a *memory operation* such as a load, store, memory barrier, or read-modify-write operation. Such a model can be precisely captured, for example, using I/O Automata [21] or similar formalisms.

We write a thread state transition as $(s, m) \to (s', m')$, where s is a thread state and m is the corresponding memory state. An *execution* is a sequence of interleaved thread transitions.

Assume we are given a *base* system $\mathcal{B}$ encompassing threads $T_0, \ldots, T_n$ and memory M. From $\mathcal{B}$ we derive a *teleported* system $\mathcal{T}$ encompassing the same number of threads $T'_0, \ldots, T'_n$ and memory M. A map f carries each state of T'_i to a state of T_i such that

- every transition in $\mathcal{B}$ has a matching transition in $\mathcal{T}$: for every transition $(s, m) \to (s', m')$ of $\mathcal{B}$ there is a transition $(\hat{s}, m) \to (\hat{s}', m')$ of $\mathcal{T}$ such that $f(\hat{s}) = s$ and $f(\hat{s}') = s'$.
- every transition of $\mathcal{T}$ corresponds to some atomic sequence of transitions of $\mathcal{B}$: if $(s, m) \to (s', m')$ is a transition of $\mathcal{T}$, then there is a sequence of transitions of $\mathcal{B}$, $(s_i, m_i) \to (s_{i+1}, m_{i+1})$, $0 \leq i \leq k$, such that $f(s) = s_0, m = m_0$ and $f(s') = s_k, m' = m_k$.

From these conditions, it is not difficult to show that every execution of the teleported system $\mathcal{T}$ maps to an execution of the base system $\mathcal{B}$.

Informally, this model just says that checking correctness for a teleportation design requires checking that the thread and memory states in $\mathcal{T}$ after teleportation match the final memory states after an uninterrupted single-thread execution of $\mathcal{B}$.

6 Related Work

As noted, *Hardware lock elision* (HLE) [25, 26] is an early example of the teleportation pattern. In HLE, a critical section is first executed speculatively, as a hardware transaction, and if the speculation fails, it is re-executed using a lock.

Another prior example of teleportation is *telescoping*, proposed by Dragojevic et al. [11] Here, hardware transactions are used to traverse a reference-counted list without the cost of incrementing and decrementing intermediate reference counts.

StackTrack [2] (arguably) incorporates the teleportation pattern within a more complex design. It provides a compiler-based alternative to hazard pointers, in which hardware transactions are overlaid on the basic blocks of non-transactional programs. On commit, each such transaction publishes a snapshot of its register contents and local variables, effectively announcing which memory locations that thread may be referencing.

There are a number of related techniques that resemble our use of teleportation in some aspects, but either do not retrofit a viable prior design, or do not exist to minimize memory traffic.

Makreshanski *et al.* [22] examine (among other issues) the use of hardware transactions to simplify the conceptual complexity of lock-free B-trees by judicious introduction of multi-word CAS operations. Here, the speculative path executes the multi-word CAS as a hardware transaction, while the non-speculative fallback path uses a complex locking protocol. Although this does not really retrofit speculative short-cuts to prior design, and the overall goal is simplicity, not efficiency gain by reducing memory traffic.

BulkCompiler [1] proposes a novel hardware architecture where a compiler automatically parcels Java applications into hardware transactions (called "chunks"). Lock acquisition and release instructions within chunks can be replaced by memory reads and writes. Instead of retrofitting an existing design, BulkCompiler requires specialized hardware and a specialized compiler, while teleportation targets commodity platforms and compilers. BulkCompiler is monolithic, operating automatically on entire applications.

Hand-over-hand locking, also known as lock coupling, is due to Bayer and Schkolnick [4]. Lock-free linked-list algorithms are due to Valois [28], Harris [14], and Heller *et al.* [16].

Hazard pointers are due to Michael [24]. Other approaches to memory management for highly-concurrent data structures include *pass the buck* [17], *drop the anchor* [5], epoch-based techniques [14], and reference counting [10, 13, 27]. Hart *et al.* [15] give a comprehensive survey of the costs associated with these approaches.

Several recent studies aimed at reducing the costs of hazard pointers. The *Optimistic Access* scheme [9, 8] reduces the cost of memory barriers and hazard pointers by letting the threads optimistically read memory without setting hazard pointers. This scheme provides a mechanism that makes a thread aware of reading reclaimed memory and allow it to restart its operation. The optimistic access scheme can be applied automatically in a garbage-collection-like manner, and it fully preserves lock-freedom. *Debra* [6] extends epoch-based reclamation by interrupting delayed threads and making them restart. Optimistic access and Debra require the algorithm to be presented in a special form that allows restarting the operation when needed.

Threadscan [3] mitigates the cost of hazard pointers by avoiding memory barriers and interrupting all threads to gather their hazard pointers or their local pointers when needed. Threadscan requires substantial support from the runtime system and the compiler. The teleportation scheme presented in this paper does not require any special algorithmic form and it does not require compiler support.

Read-copy-update [23] is a memory-management mechanism optimized for low update rates, but does not permit fine-grained concurrency among updating threads.

7 Conclusion

State teleportation exploits best-effort hardware transactional memory (HTM) to improve of memory management for highly-concurrent data structures. Because modern HTMs do not provide progress guarantees, teleportation is designed to degrade gracefully if transaction commit rates decline, either because of contention or because the platform does not support HTM. In the limit, the data structures revert to their underlying memory management schemes.

In this paper we illustrated the use of teleportation for improving the performance of various data structures, including lock-based and lock-free lists, skiplists, and trees. Teleportation successfully lowers the costs associated with multiple lock acquisition and with memory barriers required by hazard pointers.

Naturally, there are limitations to be addressed by future work. Teleportation, like hazard pointers, must currently be applied by hand, just like the underlying locking or hazard pointers mechanisms. A future challenge is to automate this process.

Acknowledgements. The authors are grateful to Tim Harris for comments.

References

1 Wonsun Ahn, Shanxiang Qi, M. Nicolaides, Josep Torrellas, Jae-Woo Lee, Xing Fang, Samuel P. Midkiff, and David C. Wong. Bulkcompiler: high-performance sequential consistency through cooperative compiler and hardware support. In David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez, editors, *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 133–144. ACM, 2009. doi:10.1145/1669112.1669131.

2 Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: an automated transactional approach to concurrent memory reclamation. In Dick

C. A. Bulterman, Herbert Bos, Antony I. T. Rowstron, and Peter Druschel, editors, *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 25:1–25:14. ACM, 2014. doi:10.1145/2592798.2592808.

3 Dan Alistarh, William M Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *SPAA*, pages 123–132, 2015.

4 R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

5 Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In Guy E. Blelloch and Berthold Vöcking, editors, *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 33–42. ACM, 2013. doi:10.1145/2486159.2486184.

6 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *PODC*, PODC '15, pages 261–270, 2015.

7 Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Q. Le. Robust architectural support for transactional memory in the power architecture. In Avi Mendelson, editor, *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 225–236. ACM, 2013. doi:10.1145/2485922.2485942.

8 Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *OOPSLA*, OOPSLA '15, pages 260–279, 2015.

9 Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *SPAA*, SPAA '15, pages 254–263, 2015.

10 David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *PODC*, PODC '01, pages 190–199, 2001.

11 Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *PODC*, pages 99–108, 2011.

12 Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

13 Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8):1173–1187, 2009. doi:10.1109/TPDS.2008.167.

14 Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300—-314, 2001.

15 Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007. doi:10.1016/j.jpdc.2007.04.010.

16 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.

17 Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *DISC*, DISC '02, pages 339–353, 2002.

18 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

19 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. URL: http://www.worldcat.org/isbn/0123705916.

20 Intel Corporation. Transactional Synchronization in Haswell. Retrieved from http:/http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 8 September 2012.

21 Nancy Lynch and Mark Tuttle. An Introduction to Input/Output automata. Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, nov 1988.

22 Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. In *VLDB*, volume 8, No. 11, 2015. URL: https://www.microsoft.com/en-us/research/publication/to-lock-swap-or-elide-on-the-interplay-of-hardware-transactional-memory-and-lock-free-indexing/.

23 P.E. McKenney and J.D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *PDCS*, pages 509–518, 1998.

24 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004. doi:10.1109/TPDS.2004.8.

25 R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305, 2001. URL: citeseer.nj.nec.com/rajwar01speculative.html.

26 R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, pages 5–17, 2002.

27 Hakan Sundell. Wait-free reference counting and memory management. In *IPDPS*, IPDPS '05, pages 24.2–, 2005.

28 J. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, pages 214–222, 1995. URL: http://citeseer.nj.nec.com/valois95lockfree.html.

Evacuating an Equilateral Triangle in the Face-to-Face Model*

Huda Chuangpishit[1], Saeed Mehrabi[2], Lata Narayanan[3], and Jaroslav Opatrny[4]

1 **Department of Computer Science, Concordia University, Montreal, Canada**
hoda.chuang@gmail.com
2 **School of Computer Science, Carleton University, Ottawa, Canada**
saeed.mehrabi@carleton.ca
3 **Department of Computer Science, Concordia University, Montreal, Canada**
lata@cs.concordia.ca
4 **Department of Computer Science, Concordia University, Montreal, Canada**
opatrny@cs.concordia.ca

Abstract

Consider k robots initially located at the centroid of an equilateral triangle T of sides of length one. The goal of the robots is to *evacuate* T through an exit at an unknown location on the boundary of T. Each robot can move anywhere in T independently of other robots with maximum speed one. The objective is to minimize the *evacuation time*, which is defined as the time required for *all* k robots to reach the exit. We consider the *face-to-face* communication model for the robots: a robot can communicate with another robot only when they meet in T.

In this paper, we give upper and lower bounds for the face-to-face evacuation time by k robots. We show that for any k, any algorithm for evacuating $k \geq 1$ robots from T requires at least $\sqrt{3}$ time. This bound is asymptotically optimal, as we show that a straightforward strategy of evacuation by k robots gives an upper bound of $\sqrt{3} + 3/k$. For $k = 3, 4, 5, 6$, we show significant improvements on the obvious upper bound by giving algorithms with evacuation times of 2.0887, 1.9816, 1.876, and 1.827, respectively. For $k = 2$ robots, we give a lower bound of $1 + 2/\sqrt{3} \approx 2.154$, and an algorithm with upper bound of 2.3367 on the evacuation time.

1998 ACM Subject Classification I.1.2. Algorithms, D.1.3 Distributed programming

Keywords and phrases Distributed algorithms, Robots evacuation, Face-to-face communication, Equilateral triangle

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.11

1 Introduction

Searching for an object at an unknown location in a specific domain in the plane is a well-studied problem in theoretical computer science [1, 4, 5, 21, 22]. The problem was initially studied when there is only one searcher, whom we refer to as a *robot*. The target is assumed to be a point in the domain, and the robot can only find the target when it visits that point. The goal then is to design a trajectory for the robot that finds the target as soon as possible. Recent work has focused more on *parallel* search by several robots, which can reduce the

* The work of LN and JO is supported in part by NSERC. This work was done when SM was visiting Concordia University.

search time as the robots can distribute the search effort among themselves. The *search time* by k robots is generally defined to be the time the *first* robot reaches the target.

A natural generalization of the parallel search problem, called the *evacuation problem*, was recently proposed: consider several robots inside a region that has a single *exit* at an unknown location on its boundary. All robots need to reach the exit, i.e., *evacuate* the region, as soon as possible. This is essentially the parallel search problem where the exit is the search target, however we are interested in minimizing the time the *last* robot arrives at the exit. The problem was introduced in [8] (see also [10]), and a number of papers on this problem have been published since then.

The evacuation problem substantially depends on the way robots can communicate among themselves. Two models of communication have been proposed: in the *wireless model*, each robot can communicate wirelessly with the other robots instantaneously, regardless of their locations. In the *face-to-face model*, two robots can communicate with each other only when they meet, i.e., when they occupy the same location at the same time. Since in the wireless model robots can communicate with each other regardless of their locations, as soon as a robot finds the exit, it can announce it to other robots. This is not possible in the face-to-face communication, which makes the evacuation problem more challenging, due to the very limited communication capabilities of robots.

In this paper, we study the problem of evacuating a unit equilateral triangle in the face-to-face model with k robots, all of which are initially located at the centroid of the triangle. Our objective is to design the trajectories of the robots so as to minimize the *worst-case evacuation time*, which is defined as the time it takes for *all the robots* to reach the exit.

Related work. A classical problem related to our paper is the well-known *cow-path* problem introduced by A. Beck [4], in which a cow searches for a hole in an infinite linear fence. An optimal deterministic algorithm for this problem and for its generalization to several fences is known, e.g., Baeza-Yates et al. [2]. Since then several variants of the problem have been studied [3, 6, 7, 11, 15, 16, 18, 17, 20].

Lopez-Ortiz and Sweet [20] asked for the worst-case trajectories of a set of robots searching *in parallel* for a target point at an unknown location in the plane. Feinerman et al. [14] (see also [13]) introduced a similar problem in which a set of robots that are located at a cell of an infinite grid and being controlled by a Turing machine (with no space constraints) need to find the target at a hidden location in the grid. In these two models of multi-robot searching, the robots cannot communicate at all. By controlling each robot by asynchronous finite state machine, Emek et al. [12] studied this problem in which the robot can have a "local" communication in some sense and proved that the collaboration performance of the robots remains the same, even if they possess a constant-size memory. Lenzen et al. [19] extended this problem by introducing the *selection complexity* measure as another factor in addition to studying the time complexity of the problem.

The evacuation problem with several robots has been studied in recent years under wireless and face-to-face models of communications. For the wireless model, Czyzowicz et al. [8] studied the problem of evacuating a unit disk, starting at the center of the disk. They gave a tight bound of 4.83 for the evacuation time of $k = 2$ robots, as well as upper and lower bounds of, respectively, 4.22 and 4.159 for $k = 3$. These bounds for k robots become $3 + \pi/k + O(k^{-4/3})$ and $3 + \pi/k$, respectively [8]. Czyzowicz et al. [11] also studied the evacuation problem for k robots for unit-side squares and equilateral triangles in the wireless model. For a unit-side square, they gave optimal algorithms for evacuating $k = 2$

robots when located at the boundary of the square. Moreover, for an equilateral triangle, they gave optimal evacuation algorithms for $k = 2$ robots in any initial position on the boundary or inside the triangle. They also showed that increasing the number of robots cannot improve the evacuation time when the starting position is on the boundary, but three robots can improve the evacuation time when the starting position is the centroid of the triangle. Recently, Brandt et al. [6] considered the evacuation problem for k robots on m concurrent rays under the wireless model. Finally, the evacuation problem on a disk with three robots at most one of which is *faulty* was recently studied by Czyzowicz et al. [9] under the wireless model.

For the face-to-face model, Czyzowicz et al. [8] gave upper and lower bounds of, respectively, 5.74 and 5.199 for the evacuation time of $k = 2$ robots initially located at the center of a unit disk. Both the upper and lower bounds were improved by Czyzowicz et al. [10] to 5.628 and 5.255, respectively. Closing this gap remains open. When $k = 3$ the upper and lower bounds for the face-to-face model are 5.09 and 5.514, respectively, and $3 + 2\pi/k$ and $3 + 2\pi/k - O(k^{-2})$ for any $k > 3$ [8].

Our results and organization. In this paper, we study the evacuation of k robots from an equilateral triangle under the face-to-face model. We present the following results:

- For $k \geq 3$, we show that any algorithm for evacuating k robots from triangle T requires at least $\sqrt{3}$ time. We prove that this bound is asymptotically optimal by giving a simple algorithm that achieves an upper bound of $\sqrt{3} + 3/k$.
- We show that a significant improvement on the above upper bound can be obtained using the *Equal-Travel Early-Meeting* strategy. In this strategy the travel time of all robots is the same and they use a *meeting point* for all robots before the entire boundary is explored to share information. Applying this strategy we design algorithms for k =2,3,4,5, and 6 with evacuation times of $\approx$ 2.4114, 2.0887, 1.982, 1.8760 and 1.823, respectively.
- For $k = 2$ we prove a lower bound of 2.154 on the evacuation time. We improve the evacuation algorithm for $k = 2$ even further by replacing an early meeting with one or several *detours* inside the triangle which improves the evacuation time to 2.3367.

We specify some preliminaries and notation in Section 2. Then, we give the proofs of our lower bounds in Section 3, and present our evacuation algorithms in Section 4. We conclude the paper with a summary of our results and a discussion of open problems in Section 5.

2 Preliminaries

For two points p and q in the plane, we denote the line segment connecting p and q by pq and the length of pq by $|pq|$. Throughout the paper, we denote an equilateral triangle by T and denote the vertices of T by A, B, and C. Thus we sometimes write ABC to refer to T. We always assume that the sides of T have length 1 and every robot moves at maximum speed 1. Throughout the paper we use the following triangle terminology:

- By h we denote the *height* of the equilateral triangle. Observe that $h = \sqrt{3}/2$.
- We denote by O the *centroid* of T (i.e., the intersection point of the three heights of T).
- We use x, and y to denote the distance of O to a vertex, and to the side of the triangle, respectively; notice that $x = 2h/3 = \sqrt{3}/3$ and $y = h/3 = x/2 = \sqrt{3}/6$.

We define $E_{\mathcal{A}}(T, k)$ to be the worst-case evacuation time of the unit-sided equilateral triangle T by k robots using algorithm $\mathcal{A}$, assuming the robots are initially located at the centroid of the triangle, the exit is located at an unknown location on the boundary of the triangle, and

the robots communicate using the face-to-face model. Also, we define $E^*(T,k)$ to be the *optimal* evacuation time of the triangle by k robots in the face-to-face model.

A deterministic algorithm for the evacuation problem by k robots takes as input the triangle and the k robots located at its centroid, and outputs for each robot a *fixed trajectory* consisting of a sequence of connected line segments or curves to be followed. We assume every robot knows the trajectories of all the robots. A robot R follows its trajectory unless:

- R sees the exit: R may then quit its trajectory and go to a point where it can intercept another robot and inform it about the exit.
- R meets another robot who has found the exit: R then quits its trajectory and proceeds directly to the exit.

Observe that the robots are initially co-located, and the initial part of their trajectories may be identical, i.e., when going to the boundary of the triangle. Later on, the trajectories of two or several robots may intersect and the intersection point may be reached by all robots at the same time. We call such a point a *meeting point*. A meeting point might be in the interior of the triangle and it can serve as a place for the robots to exchange information about the progress in the search for exit. If one of the robots has found the exit, they can proceed towards it. Otherwise, the robots can continue in the search for the exit separately or together. As shown in [11], and in Section 4, an algorithm with a meeting point in the interior of the region can improve the evacuation time in some cases.

If the trajectory of a robot leaves the boundary of the triangle and returns to the boundary without a planned meeting point, we say that the robot makes a *detour*. The robot may make such a detour to enable another robot that has found the exit to intercept it. In the absence of an interception, the robot has gained information about the *absence* of the exit in some part of the boundary. A detour in the trajectories of two robots was used in [10] to improve the evacuation time, and we use this strategy in Subsection 4.2.

3 Lower Bounds

In this section, we prove lower bounds on the evacuation time. We first show that regardless of the number of robots, $\sqrt{3}$ is a lower bound on the evacuation time. This bound holds even if the exit is known to be at one of the three vertices of the triangle.

▶ **Theorem 1.** *Consider k robots $R_1, R_2, \ldots, R_k$, initially located at the centroid of an equilateral triangle T. In the face-to-face model the evacuation time of k robots $E^*(T,k) \geq \sqrt{3} \approx 1.732$.*

Proof. Consider a deterministic and arbitrary evacuation algorithm $\mathcal{A}$ for k robots. We first run the algorithm to see which vertex is the last one visited by the robots (two or even three vertices could be visited at the same time as the last ones in which case, we choose an arbitrary one as last). Assume without loss of generality that A is the last vertex visited by any of the robots; let I_1 be the input in which the exit is at A. Consider the execution of the algorithm on input I_1, and let t be the time the second of the three vertices is visited by some robot R. Without loss of generality, let this second vertex be B; that is, R visits vertex B at time t on input I_1. Let I_2 be the input where the exit is at the remaining vertex C. We argue that the evacuation time of the algorithm must be $\geq 3x$ on one of these two inputs.

If $t \geq 3x-1$, then it takes an additional time 1 for robot R to reach the exit at A, leading to a total evacuation time of at least $3x$ on input I_1. Therefore, assume that $t < 3x-1$. Since R has to reach B before time $3x-1$, we claim that it is impossible for R to meet a robot R' that has already visited A or C before R reaches B at time t. Suppose R was

able to meet R' that had visited A (without loss of generality) at some meeting point M at time t_M. Then clearly $t_M \geq x + |AM|$. After meeting R', the robot R needs time at least $|MB|$ to get to B. We conclude that $t \geq t_M + |MB| \geq x + |MB| + |MA| \geq x + 1$. However, $x + 1 > 3x - 1$, a contradiction. Thus, R's trajectory to B, reaching B at time $t < 3x - 1$ cannot allow a meeting between R and any robot that has already visited A or C. Therefore, the behaviour of the robot R must be the same on inputs I_1 and I_2 until time t when R arrives at B. Observe now that $t \geq x$. At time $2x$, if the robot R is at distance $> x$ from A, the adversary puts the exit at A (input I_1), and if it is at distance $> x$ from C, it puts the exit at C. Combined with the fact that at time $2x$, the robot R can travel at most distance $2x - t \leq x$ from B, we have the desired result. ◀

The above bound is asymptotically optimal, as we will describe a simple algorithm in Section 4 that evacuates k robots in $\sqrt{3} + 3/k$ time from an exit situated anywhere on the boundary. We remark also that in the wireless communication model, $E^*(T, 6) = \frac{2\sqrt{3}}{3}$ (D. Krizanc, private communication, 2015), showing that for the equilateral triangle, evacuation even with arbitrarily many robots takes much more time in the face-to-face model, than evacuating only six robots in the wireless model.

When $k = 2$, we are able to prove a better lower bound of ≈ 2.15. The argument used for the lower bound is an adversary argument: depending on what the algorithm does, the adversary places the exit in such a way so as to force the claimed evacuation time. The key insight can be summarized as follows: if an algorithm is to do better than the claimed lower bound, either the robots cannot meet in a useful way to shorten the time to reach the exit, or they simply cannot finish the exploration. To this end, we first prove the following technical lemma. We first need some notation. For the equilateral triangle ABC, let D, E and F denote the middle point of sides AB, AC and BC, respectively, and let $\mathcal{S} = \{A, B, C, D, E, F\}$. We say two points in $\mathcal{S}$ have *opposite positions* if one point is a vertex of the triangle T and the other point is located on the opposite side of that vertex. For example, the vertex C and a point in $\{A, D, B\}$ have opposite positions.

▶ **Lemma 2** (Meeting Lemma). *Consider a deterministic algorithm $\mathcal{A}$ for evacuating two robots that are initially at the centroid of an equilateral triangle T, and let $p_1, p_2 \in \mathcal{S}$ have opposite positions. If $\mathcal{A}$ specifies a trajectory for one of the robots in which it visits p_1 at time t' satisfying $t' \geq 0.5 + y$ and a trajectory for the other robot in which it visits p_2 at time t such that $t' < t < 0.5 + h + y = 0.5 + 4y$, then the two robots cannot meet between time t and t'.*

Proof. Suppose for a contradiction that the robots meet at time $t' < t_m \leq t$ at some point z. Since p_1 and p_2 have opposite positions $|p_1p_2| \geq h$. Therefore, $|p_1z| + |zp_2| \geq h$. Moreover $|p_1z| \leq t_m - t'$ and $|zp_2| \leq t - t_m$. This implies that

$$h \leq |p_1z| + |zp_2| \leq (t_m - t') + (t - t_m) = t - t' < 0.5 + h + y - t' \leq 0.5 + h + y - 0.5 - y = h,$$

which is a contradiction. ◀

▶ **Theorem 3.** *Consider 2 robots R_1, R_2, initially located at the centroid of an equilateral triangle T. If the robots communicate using face-to-face model, then the evacuation time of two robots $E^*(T, 2) \geq 1 + 4y = 1 + 2/\sqrt{3}$.*

Proof. Suppose for the purpose of contradiction, that there is a deterministic algorithm $\mathcal{A}$ for evacuation by two robots, such that $E_{\mathcal{A}}(T, 2) < 1 + 4y$. We first focus attention on the set of points $\mathcal{S} = \{A, B, C, D, E, F\}$. There exists some input I on which the exit is the last

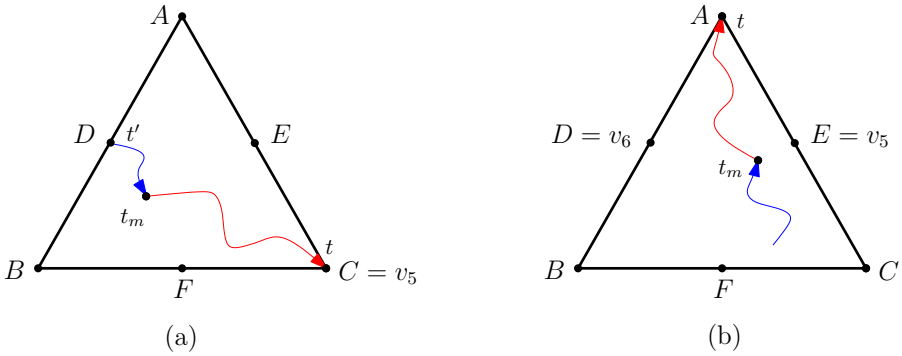


Figure 1 (a) An illustration in support of the proof of the Meeting Lemma. (b) An illustration in support of case 2 in the proof of Theorem 3.

point in $\mathcal{S}$ to be visited by either of the robots, according to the trajectories specified by $\mathcal{A}$. Let t be the time the *fifth* point of $\mathcal{S}$ is visited by a robot on input I. Let $v_1, v_2, \ldots, v_6$ be the order in which the points in M are visited by the robots, on input I; the exit is at v_6. Without loss of generality assume that v_5 is visited by robot R_1. Clearly, v_6 is not yet visited before time t; it may be visited at or after time t. First, note that since at least five points are visited at or before time t, one of the robots must have visited at least three points in M. It follows that $t \geq 1 + y$. If $t \geq 0.5 + 4y$, since the exit is at v_6, which is at least 0.5 away from R_1, we obtain $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$, a contradiction. We conclude that $1 + y \leq t < 0.5 + 4y$.

We now consider the following exhaustive cases depending on whether v_5 is a vertex of T or a midpoint of a side of T.

Case 1. v_5 is a vertex of T. Without loss of generality assume that v_5 is C. See Figure 1(a). If v_6 is any of A, D, B, then at time t, R_1 needs time at least h to arrive to v_6, which implies that $E^*(T, 2) \geq t + h \geq 1 + 4y$, a contradiction. So we conclude that v_6 is at either E or F. Since $t < 0.5 + 4y$, robot R_1 could have visited at most one of A, D, B by time t. This means that R_2 must have visited at least two of A, D, B. Let v be the second vertex of the set A, D, B to be visited by R_2, and assume it arrives there at time t'. Note that $t' \geq 0.5 + y$. By the Meeting Lemma, the two robots do not meet at any time between t' and t on input I.

Now consider an input I' in which the exit is at v. Clearly the robots behave identically on both inputs I and I' until time t'. After this time, R_2 on seeing the exit at v may behave differently; however robot R_1 must behave exactly as in I unless it meets robot R_2, which by the Meeting Lemma, cannot happen until time t. Therefore, after time t, it takes at least an additional h to reach the exit at v, giving a total evacuation time of at least $t + h \geq 1 + 4y$, a contradiction.

Case 2. v_5 is a midpoint of a side of T, and v_6 is another midpoint: Without loss of generality assume that v_5 is E, and v_6 is D; see Figure 1(b). Then, all three vertices must have been visited before or at time t. Since R_1 cannot visit two vertices *before* arriving at E at time $t < 0.5 + h + y$, we conclude that R_2 must visit two vertices by time t. Referring to Figure 1(b), consider the second vertex visited by R_2. Observe that R_2 cannot arrive there before time $1 + x$. (i) If it is B, then we put the exit at E. This way, R_2 needs time at least h to get to E from B and so $E^*(T, 2) \geq 1 + x + h \geq 1 + 4y$. (ii) If it is C, then we put the exit at D. Then, R_2 needs time at least h to get to D from C and so $E^*(T, 2) \geq 1 + x + h \geq 1 + 4y$.

We conclude that the second vertex visited by R_2 must be A. We first note that R_2 cannot have visited all F, B, A or all F, C, A by time t, as visiting either set of three points takes time at least $0.5 + x + h > 0.5 + y + h > t$. So, R_1 must have visited F and C (or F

and B) before coming to E. Let P be the second of the two points visited by R_1 before going to E. Note that P could be F, C, or B, and suppose R_1 visits P at time t'. Clearly, $t' \geq 0.5 + y$. Since R_2 must visit A at or before time $t < 0.5 + 4y$, by the Meeting Lemma, the robots cannot meet at any time $t' \leq t_m \leq t$. In other words, R_1 cannot meet R_2 after the latter has visited P and reach A on time. Now consider the input I' in which the exit is at P. On input I', the robot R_2 will have the same behavior as input I until it reaches A at time $t > 1 + x$ and then needs to get to the exit (which is at F or C or B). Therefore we have $E^*(T, 2) \geq 1 + x + h \geq 1 + 4y$, a contradiction.

Case 3. v_5 is a midpoint of a side of T, and v_6 is a vertex: Without loss of generality assume that v_5 is E. If v_6 is B, then R_1 needs time $\geq h$ to reach the exit, so on input I, the evacuation time is at least $t + h \geq 1 + h + y$. Therefore, assume without loss of generality that v_6 is A; see Figure 1(b). If a single robot visits both B and C, it takes time at least $2 + x > 1 + 4y$ to reach vertex A. Therefore, B and C must be visited by different robots. We consider separately the two cases: R_1 visits C and R_2 visits B; and R_1 visits B and R_2 visits C.

Suppose R_1 visits C before visiting E, and R_2 visits B. First observe that R_1 cannot also visit D, as visiting C, D, and E takes time at least $0.5 + 4y$, a contradiction to $t_E < 0.5 + 4y$. Therefore R_2 must visit D in addition to B. Either R_1 or R_2 must visit F. If R_2 visits F, Lemma 5 assures that $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$ and if R_2 does not visit F, Lemma 6 does the same.

Suppose instead that R_1 visits B before visiting E and R_2 visits C. Then R_1 cannot visit both D and F, as visiting D, B, F, E takes time at least $1.5 + y > 0.5 + 4y$, Lemmas 7, 8, and 9 now assure that $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$ for the cases when R_1, in addition to visiting B and E, visits F, visits D, and visits neither, respectively. ◀

For all the lemmas below, we assume that according to algorithm $\mathcal{A}$, we have $v_5 = E$ and $v_6 = A$, and robot R_1 visits E at time $1 + y \leq t_E < 0.5 + 4y$. We start with a simple observation that is used repeatedly.

▶ **Observation 4.** *Let p be a point on the boundary. If at time $1 + 4y - |Ap|$, both A and p are unvisited then $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$.*

Proof. Put the exit at whichever of the two points is visited later. Since at time $1+4y-|Ap|$, neither is visited, the time to evacuate is at least $1 + 4y - |Ap| + |Ap| = 1 + 4y$. ◀

▶ **Lemma 5.** *If R_2 visits B, D and F, and R_1 visits C and E, then $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$.*

Proof. First, observe that if B is not visited *first* of the three points B, D, F, then $t_B > 0.5+y$. Since E is visited by R_1 and $t_E \geq 1 + y$, by the Meeting Lemma, R_1 and R_2 cannot meet between t_B and t_E. Thus if the exit is at B, it will take R_1 time at least $t_E + h \geq 1 + 4y$ to reach there. We conclude that B must be visited first. If R_2 visits B, D, and F in that order, then $t_F \geq 1 + x$, so by Observation 4, we have $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$. So, R_2 must visit B, F and D in this order.

Let P be the closest point from B on the BD line segment that is not visited by R_2 before it visits F. Then the time for R_2 to reach F is at least $|OP|+|PB|+|BF|$ (Figure 2, the blue trajectory). Therefore, the earliest time R_2 can reach P is $|OP|+|PB|+|BF|+|FP|$. It can be verified that for any point M on the BD line segment, this time is more than $1+4y-|AM|$, therefore it is true for the point P defined above. Also, R_1 cannot visit P on time: if it visits P before C (Figure 2, the green trajectory), we have $t_E \geq |OP| + |PC| + |CE| \geq |OD| + |DC| + |CE| = 0.5 + 4y$, and if it visits C before P (Figure 2, the red trajectory), we have $t_E \geq |OC| + |CP| + |PE| \geq |OC| + |CD| + |DE| = 2y + 3y + 0.5$. Thus neither robot can visit P before time $1 + 4y - |AP|$. The lemma now follows from Observation 4. ◀

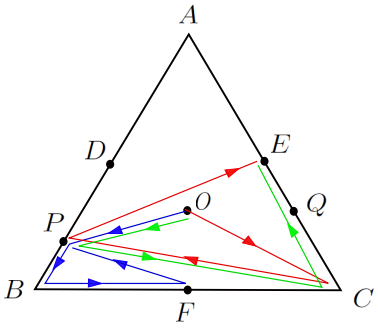


Figure 2 An illustration of possible trajectories of R_1 and R_2, in support of Lemma 5.

▶ **Lemma 6.** *If R_2 visits B and D and R_1 visits C, F, and E, then $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$.*

Proof. First observe that if R_2 visits D before B, then $t_B \geq 0.5 + y$. Since E needs to be visited by the other robot R_1 and $t_E \geq 1 + y$, by the Meeting Lemma, R_1 and R_2 cannot meet between t_B and t_E. Thus if the exit is at B, it will take R_1 time at least $t_E + h \geq 1 + 4y$ to reach there. We conclude that B must be visited first. Also, clearly E is visited last by R_1. If R_1 visits F before C, then $t_C \geq 0.5 + y$, and $t_D \leq t_E \leq 0.5 + 4y$, so by the Meeting Lemma, R_1 and R_2 cannot meet before time $0.5 + 4y$ and after R_2 visits B. Thus if the exit is at B, it will take R_1 time at least $t_E + h \geq 1 + 4y$ to reach there, and we conclude that $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$. Therefore, R_1 must visit C, F, and E in that order. Using a similar argument as in Lemma 5, we can see that there exists an unvisited point P on the CE segment at time $1 + 4y - |AP|$. It follows from Observation 4 that $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$. ◀

▶ **Lemma 7.** *If robot R_1 visits B, F and E, and R_2 visits C and D, then $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$.*

Proof. First observe that E must be visited last, and if R_1 visits F before B then $t_E \geq 0.5 + 4y$. So B must be visited before F. Now let P be a point at distance 0.3 from B on the BD segment, and Q a point at distance 0.34 from C on the CE segment. It can be verified that if R_1 visits a point on the PD segment before arriving at B, then $t_E \geq 0.5 + 4y$. Similarly, R_1 cannot visit any point in the QC line segment if it is to reach E by time $0.5 + 4y$. See Figure 3 (a). Therefore the entire PD line segment and the entire QC line segments must be visited by R_2. Now we consider the order of visiting D, Q, C, P. If D or P are visited before C, then C cannot be reached before $4y$ which means that if the exit is at A, R_2 cannot reach it before time $1 + 4y$. So C has to be visited before P or D, Figure 3 (b). Regardless of whether Q or C is visited first, it can be verified that it is impossible for R_2 to reach P before time $1 + 4y - |AP|$, yielding the desired conclusion, using Observation 4. ◀

▶ **Lemma 8.** *If robot R_1 visits B, D and E, and R_2 visits C and F, then $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$.*

Proof. R_2 must visit F before C, as otherwise as shown in the proof of Lemma 5, there will be a point P on the CE segment that cannot be visited before time $1 + 4y - |AP|$. If R_1 visits D before B, then $t_E \geq 0.5 + 4y$. So R_1 must visit B before D. By Observation 4, the entire BC edge must be visited at or before time $1 + y$. Let Q be the leftmost point on the BC edge that is not visited by robot R_1. Then R_2 must visit the entire QC segment. Since R_2 must visit C before time $4y$, we see that $|BQ| > 0.236$. As a result $t_D \geq 1.13155$. Let R be the point at distance 0.05 from D on the the DA segment, and S be the point at distance

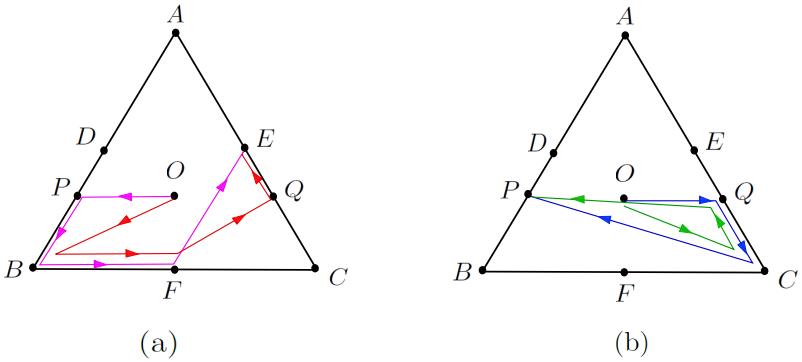


Figure 3 (a) An illustration of possible trajectories of R_1 and (b) possible trajectories of R_2, in support of Lemma 7.

0.03 from E on the EC segment. Using the assumption that it reaches E before time $0.5+4y$, it can be verified that R_1 cannot visit either R or S before time $1.657 > 0.5 + 4y$, and $1.661 > 0.5 + 4y$ respectively. Then since the SE segment must be visited by R_2 before time $0.5 + 4y$, R_2 cannot reach R before time $|OF| + |FC| + |CS| + |SR| > 1.75 > 1 + 4y - |AR|$. It follows from Observation 4 that $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$. ◀

▶ **Lemma 9.** *If robot R_1 visits B and E, and R_2 visits C, F, and D, then $E_{\mathcal{A}}(T, 2) \geq 1 + 4y$.*

Proof. We observe that D must be visited last; if F is visited last, $t_F \geq 0.5 + 4y$, and if C is visited last, then $t_C \geq 1 + y > 4y$. In both cases, Observation 4 gives the desired result. The rest of the proof is analogous to the case when robot R_1 visits B, F, E. ◀

4 Evacuation Algorithms and Upper Bounds

In this section, we give evacuation algorithms for k robots, $k \geq 2$, that are initially located on the centroid of an equilateral triangle, and derive upper bounds on the evacuation time by analyzing their performance.

Consider a straightforward strategy for evacuating k robots, that we call the *Equal-Exploration* strategy: divide the boundary into k equal-sized contiguous sections of length $3/k$ each, and assign each robot to explore a unique section of the boundary. Each robot goes to one endpoint of its assigned section, it explores its assigned section in time at most $3/k$, then it returns to the centroid to meet the other robots to share the result of its exploration. Since the sections can be chosen so that no section begins at a vertex, each robot takes time less than $2x$ for the total travel to and from the centroid. Thus, all robots are at the centroid at time less than $2x + 3/k$. Finally, all robots travel together to the exit taking time at most x. Clearly, this algorithm has evacuation time less than $3x + 3/k = \sqrt{3} + 3/k$. Although very simple, by Theorem 1, this bound is asymptotically optimal. As such, we have the following:

▶ **Observation 10.** *The Equal-Exploration strategy for k robots has worst case evacuation time less than $\sqrt{3} + 3/k$.*

In the rest of this section we propose several improvements of the above strategy and obtain better upper bounds for $k \leq 6$. However, our strategies can be easily used to obtain evacuation algorithms for other values of k.

4.1 Equal-Travel Early-Meeting Algorithms

First notice that the time to travel from and to the centroid can vary by almost a factor of 2 for different robots. In particular, robots that are assigned a section of the boundary starting or ending close to a midpoint have to travel a much smaller distance to or from the centroid, than robots that are assigned a section of the boundary that starts/ends close to a vertex. In the worst case the exit could be discovered by a robot that is the last to arrive at the centroid. Thus, our second strategy, which we call the *Equal-Travel* strategy, divides the boundary into sections in such a way that the total lengths of trajectories of all robots is equalized. In other words, we aim to equalize the *travel time* of all robots. Clearly, the Equal-Travel strategy should in general lower the evacuation time compared to the Equal-Exploration strategy. For a general value of k, such a division into equal length trajectories requires a solution of a system of equations of order 4, without a significant improvement for large values of k, and thus we did not do it in general.

Second notice that in both the Equal-Exploration and Equal-Travel strategies, the robots meet in the interior of the triangle *after* the entire boundary has been explored, and then travel together to the exit. In the following we show that the evacuation time of a triangle can be improved if the robots stop the exploration of the boundary *early* and go to a meeting point before the boundary is explored entirely. After this *early meeting*, either the robots go together to the exit, or they all go together to explore the rest of the boundary where the exit is now known to be.

In the sequel, we describe Equal-Travel Early-Meeting algorithms that combine the equal travel strategy with an early meeting. Such an algorithm for the evacuation of k robots selects the location of the early meeting point. It divides the boundary into $k+1$ sections, of which k require the same travel times, and are assigned to unique robots, and the last section is a *common section*, to be explored together by all robots. In the first phase each robot explores its assigned section (the last section remains unexplored) and returns to the meeting point. If the exit has been found by one of them, all robots proceed to the exit. If the exit has not been found, it must lie in the unexplored last section. The robots go together to explore the common last section and evacuate together. By optimizing the position of the meeting point, the location of the sections, and the length of the common section, we obtain Early-Meeting algorithms with better performance than those using only the Equal-Travel strategy. Next we give the details of the Early-Meeting algorithm for each $k \in \{2, 3, 4, 5, 6\}$.

▶ **Theorem 11.** *There are Equal-Travel Early-Meeting algorithms for two and three robots with evacuation times* $E^*(T, 2) \leq 2.4113$*, and* $E^*(T, 3) \leq 2.08872$.

Proof. First we consider the case of 2 robots. Let p be the point in the middle of side BC, and r_1, r_2 be points on sides AB, AC to be determined later, and the meeting point D be the bisector of segment r_1r_2 as on Figure 4(a). R_1 is assigned the section of the boundary from p to B to r_1, and R_2 is assigned the section from p to C to r_2, the rest of the boundary being the final common section. Now, if the exit is discovered by R_2 then R_1 travels the distance at most $t_1 = y + 0.5 + |Br_1| + |r_1D| + |DC|$. If the exit is discovered in the common section, it travels distance at most $t_2 = y + 0.5 + |Br_1| + 2|r_1D| + 2|Ar_1|$. The worst-case evacuation time is $max(t_1, t_2)$. Solving the equation $t_1 = t_2$, we obtain $|Br_1| = 0.68868$ and the evacuation time is at most 2.4113.

Next, we consider the case of 3 robots. Consider the following instance of the Early-Meeting strategy. Place points p_1, r_1, r_2 on sides BC, BA, AC as shown in Figure 4(b), the exact locations to be determined later. Centroid O is designated as the meeting point of the robots. Point p_2 is placed so that the distance $|Op_1| + |p_1p_2| = x$, the line segment p_1p_2

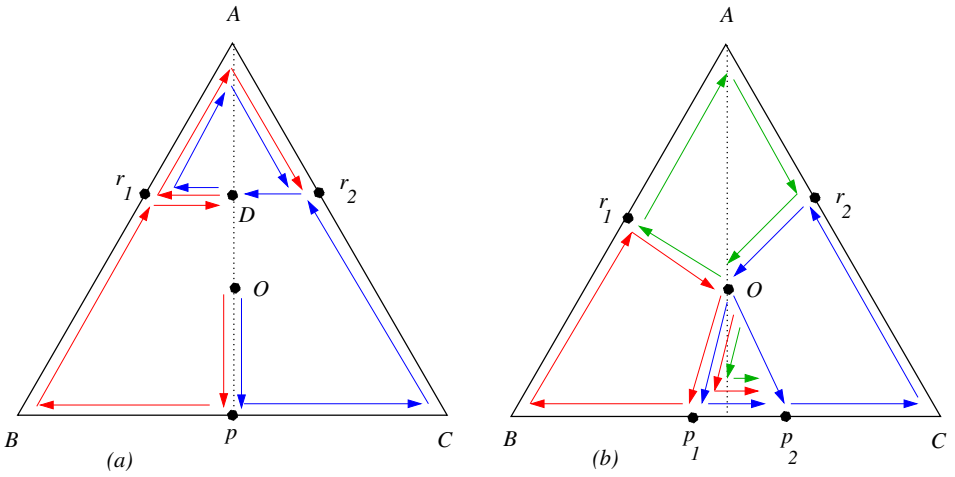


Figure 4 Equal-Travel Early-Meeting trajectories for (a) two robots, and (b) three robots.

being designated as the common section. R_1 is assigned the section of the boundary from p_1 to B and to r_1, R_2 the section from r_1 to A to r_2, and R_3 the section from r_2 to C to p_2. If one of the robots discovers the exit in its section, the other robots need to travel to it from the meeting point at the centroid, which adds distance at most x. In this setup the maximum distance robots travel are

$$t_1 = |Op_1| + |p_1B| + |Br_1| + |r_1O| + x \text{ for } R_1,$$
$$t_2 = |Op_2| + |p_2C| + |Cr_2| + |r_2O| + x \text{ for } R_2, \text{and}$$
$$t_3 = |Or_1| + |r_1A| + |Ar_2| + |r_2O| + x \text{ for } R_3.$$

Solving the set of equations $t_1 = t_2$, $t_2 = t_3$ and then optimizing the position of p_1, we get that the optimal placement of points is $|Bp_1| = .4545932809$, $|Br_1| = .4747719935$ and $|Cr_2| = .5454067191$, and the evacuation time of the algorithm at most 2.08872. All equations were solved and optimizations done using the Maplesoft [23]. ◀

It is easy to see that for $k = 2$ and $k = 3$ the Equal-Travel algorithms without any early meeting have evacuation times $y + 2.5 \approx 2.788$ and $2y + 1 + x \approx 2.155$, respectively.

We remark here that the Equal-Travel Early-Meeting algorithm for two robots given above can be further improved by additional optimization of the position of the meeting point D. We have not done it because in Section 4.2 we show a different strategy, applicable only to two robots, which gives an evacuation time that is better than that obtained with the optimized Early-Meeting strategy.

Clearly the approach used in Theorem 11 can be generalized to any $k > 2$. Start with selecting a position p_1 for the beginning of the common section and partition arbitrarily the boundary by placing points $r_1, r_2, \ldots, r_{k-1}$, see Figure 5 for $k = 4$ and 5. Point p_2 is placed so that the distance $|Op_1| + |p_1p_2| = x$. Similarly as above, we obtain a system of $k - 1$ equations for the values of $r_1, r_2, \ldots, r_{k-1}$ that produce equal travel time for all robots in the first phase. Finally, by optimizing the value of p_1 we get the final assignment of the trajectories of robots. In this manner we obtained the following theorem.

▶ **Theorem 12.** *There are Equal-Travel Early-Meeting algorithms for $k = 4$, 5, and 6 robots with $E^*(T, 4) \leq 1.9816$. $E^*(T, 5) \leq 1.876$, $E^*(T, 6) \leq 1.8263$, respectively.*

Proof. Similar to the proof of Theorem 11 and thus omitted. ◀

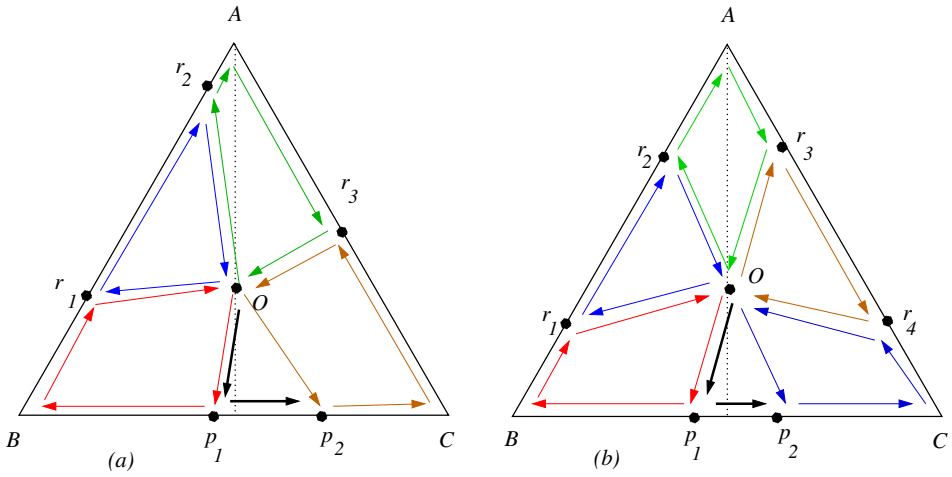


Figure 5 Establishing Equal-Travel Early-Meeting Trajectories of (a) four robots, and (b) five robots. The common part of trajectories of the robots is shown as thick black line.

As one could expect, the improvement in the evacuation time obtained by adding an early meeting to evacuation algorithms diminishes with increasing k. For 6 robots there is an Equal-Travel algorithm with the evacuation time of 1.8411, which is only 0.065 more than that of the algorithm from Theorem 12.

4.2 Equal-Travel with Detour Algorithms for Two Robots

If the evacuation problem is limited to two robots, the Equal-Travel Early-Meeting strategy can be improved further by using an *Equal-Travel with Detour* strategy in which an early meeting of two robots is replaced by a *detour*. This gives a further improvement on the evacuation time. The idea of using a detour was originally mentioned in the context of evacuating a disk with two robots in [8]. In this paper we show that a detour can also improve the evacuation time in an equilateral triangle, and multiple detours can improve it further.

As in the Equal-Travel Early-Meeting strategy, trajectories of both robots have the same length. The trajectory for each robot consists of multiple sections of the boundary, separated by detours. Each robot starts with the exploration of a section of the boundary. At some point, the robot leaves the boundary and makes a detour inside the triangle (see Figure 6 for an example). If it is not intercepted by the other robot during the detour, the robot concludes that the exit was not found by the other robot in its first section, and goes back to the boundary to explore its second section of the boundary. After the robot has finished exploring its second section of the boundary, it can make another detour, and so on. It is critical that the detours are designed so that if one of the robots finds the exit in its section, it has enough time to intercept the other robot during the corresponding detour of the other robot. We point out the salient features of the detour strategy, and its differences from the early meeting strategy:

- There are no early meeting points; the trajectories of the robots do not intersect except for the initial part to get to the boundary of the triangle, and at the very end. The detour part of trajectories of robots inside the triangle only get close to each other. This makes trajectories of the robots shorter.
- Each robot is assigned more than one section of the boundary.

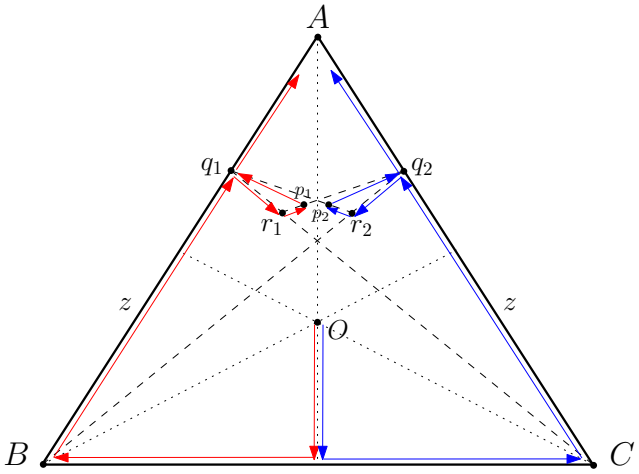


Figure 6 The trajectories of two robots with one detour for each robot.

- There is no common section of the boundary to be explored by both robots.
- A robot can do multiple detours.

We first give the details of an Equal Travel with Detour algorithm with a single detour.

▶ **Theorem 13.** *There is an Equal-Travel with Detour evacuation algorithm for two robots, using a single detour, that has evacuation time* ≤ 2.3838.

Proof. The Equal-Travel with Detour trajectories of the robots are shown in Figure 6. Points q_1 and q_2 are located symmetrically on the sides AB and AC at distance $z > 0.5$ (the exact value to be determined later) from B and C, respectively. Since the trajectories of robots are symmetric, we specify below the trajectory of R_2 only. R_2 is assigned the section from the midpoint of BC to C and from C to q_2. It starts a detour at point q_2 where it goes in the direction of point B. Point r_2 is chosen to be on the line segment q_2B so that $z + |q_2r_2| = |r_2B|$.
From r_2 the detour of R_2 goes in the direction of point q_1 until point p_2 which is chosen so that $|q_2r_2| + |r_2p_2| = |p_2q_1|$. The detour part of the trajectory of R_2 terminates with the line segment p_2q_2, where its trajectory continues with the section of the boundary from q_2 to A. Let $t_1 = y + 0.5 + z + |q_2B|$, and $t_2 = y + 0.5 + z + |q_2r_2| + |r_2p_2| + |p_2q_2| + 2(1 - z)$. We argue below that the worst case evacuation time of this algorithm is $max(t_1, t_2)$. We consider all possible locations for the exit on the sections of the boundary explored by R_1 (the case when the exit is discovered by R_2 is symmetric).

Clearly, if the exit is located on the line segment q_1A, then the evacuation time is at most t_2. If the exit is found by R_1 on the side BC, then it can intercept R_2 at point r_2 since $z + |q_2r_2| = |r_2B|$, and the robots reach the exit in time at most t_1. Assume now that R_1 finds the exit on the line segment q_1A at D, see Figure 7. Consider the triangle with sides y_1, y_2, y_3 obtained by drawing a line though point D parallel with the line going through q_1 and r_2. Since $z > 0.5$, it is easy to see that $|q_1r_2| < |Bq_1| < |Br_2|$ and by the similarity of triangles $y_3 < y_1 < y_2$. Let D' be the intersection point of the line going through q_1 and r_2 with the line drawn through D and parallel with Br_2. Since $y_1 + |DD'| < y_2 + |DD'| = |Br_2|$, when R_2 reaches r_2 robot R_1 is at D' to intercept R_2, and the distance traveled by R_2 from r_2 to D is $y_3 + |DD'| < |Br_2|$, and so the total time to reach D is less then t_1. Notice that if the value of y_1 is so that the parallel line through D would not intersect the line segment r_2p_2 then R_1 can intercept R_2 at p_2 and the the evacuation time remains less than t_1.

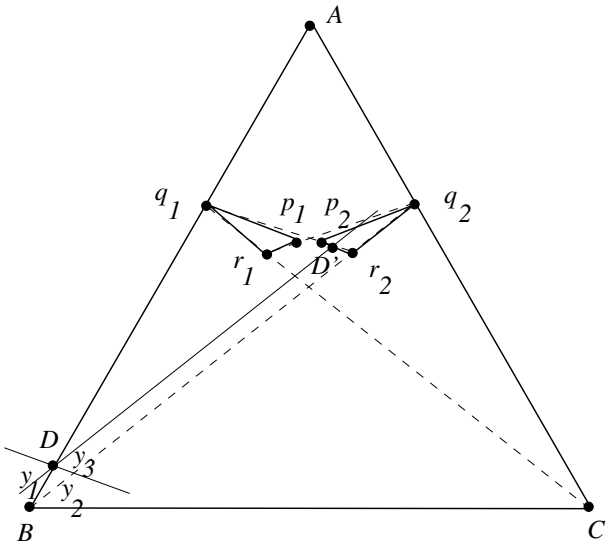


Figure 7 Evacuation time for exit at D is less that that for B.

We have shown that the worst-case evacuation time is $max(t_1, t_2)$. By equating $t_1 = t_2$, we obtain $z = 0.7151$ and $t_1 = 2.3837$. Thus the worst-case evacuation time of this algorithm is 2.3837. ◀

Consider the Equal-Travel with Detour algorithm described above and suppose that the robots do not find the exit in the first phase. That is, the robots will go back to points q_1 and q_2 to explore the rest of the boundary of T (i.e., the line segments q_1A and q_2A). Observe that for triangle $\triangle Aq_1q_2$, at this time the robots are in the same situation as they were when visiting vertices B and C for the triangle $\triangle ABC$. Furthermore, like for the triangle $\triangle ABC$, where the worst case occurs in the neighbourhood of B and C, the worst case for triangle $\triangle Aq_1q_2$ occurs when the exit is located in the neighbourhood of q_1 or q_2. Therefore, we can use an additional detour in the triangle Aq_1q_2 and improve the evacuation time in the top part as illustrated in Figure 8. However, to improve the evacuation time for the whole triangle, we need to re-balance all of the worst case evacuation times in $\triangle ABC$. In particular, we derive the expression t_1 for the maximum evacuation time if the exit is discovered by R_1 before reaching q_1, the expression t_2 for the maximum evacuation time if the exit is in the line segment q_1q_1', and the expression t_2 for the maximum evacuation time if the exit is in the line segment $q_1'A$. By solving the equations $t_1 = t_2, t_2 = t_3$ we calculated the optimized positions of q_1, q_2, q_1', q_2' using numerical calculations, obtaining $|Bq_1| = 0.666$, $Bq_1' = 0.9023$ and the evacuation time 2.3367. Thus we have the following improved evacuation time.

▶ **Theorem 14.** *There is an Equal-Travel with Detour evacuation algorithm for $k = 2$ robots that uses two detours with the evacuation time at most* 2.3367.

We remark that the use of an additional detour as above can be applied any number of times in the upper part of the triangle. With each additional detour in the upper part of the triangle we have one more case of maximal evacuation time and one more equation to add to the system of equations to be solved. However, with each additional detour the upper part of the triangle becomes much smaller, and thus the improvement in the evacuation time is extremely tiny for more than two detours.

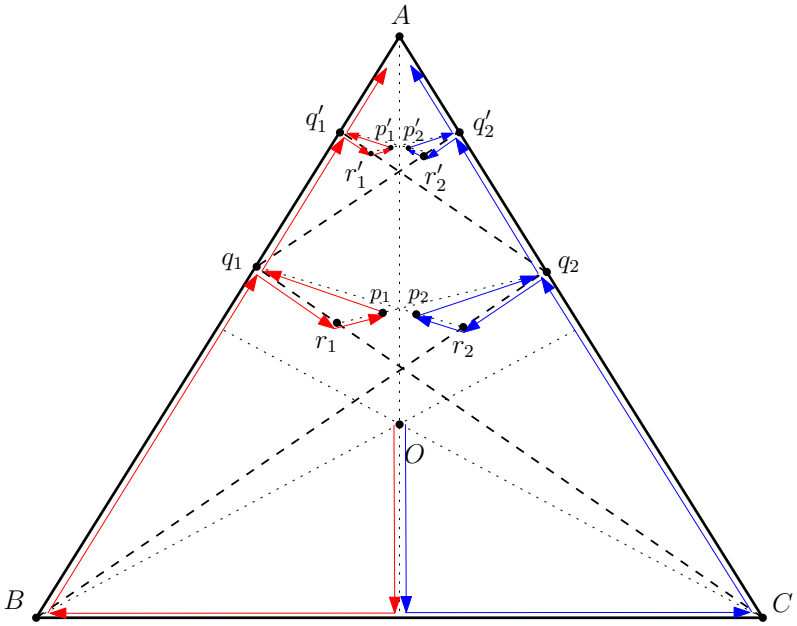


Figure 8 Trajectories of two robots with two detours.

5 Discussion

We studied the evacuation of an equilateral triangle by k robots, initially located at its centroid. The robots can communicate only if they are in the same position at the same time, i.e., they use the face-to-face communication. We showed a lower bound of $\sqrt{3}$ on the evacuation time for any number k of robots, and gave a simple strategy that achieves this bound asymptotically. We introduce the Equal-Travel Early-Meeting strategy for evacuation algorithms in which the robots meet at an early meeting point inside the triangle before the whole perimeter is examined. This strategy gave us upper bounds of 2.08872, 1.9816, 1.876 and 1.827 for $k = 3, 4, 5$, and 6 robots, respectively.

For $k = 2$ robots, we proved a lower bound of $1 + 2/\sqrt{3} \approx 2.154$. We then show that for $k = 2$ the Early-Meeting strategy can be improved by replacing the early-meeting by shorter detours in the interior of the triangle, and obtained an upper bound of 2.3367 on the evacuation time. This upper bound is achieved with two detours.

Although we limited our study to the evacuation of the equilateral triangle, the algorithmic strategies used in this paper should be applicable to other search domains. Finding tight bounds for the evacuation time in the face-to-face model remains an open problem. A clear understanding of the search domains in which early meetings or detours are *provably* useful also remains elusive.

References

1 R. Ahlswede and I. Wegener. *Search problems*. Wiley-Interscience, 1987.

2 R. A. Baeza-Yates, J. C. Culberson, and G. J. E. Rawlins. Searching with uncertainty (extended abstract). In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory (SWAT 88)*, pages 176–189, 1988.

3 R. A. Baeza-Yates and R. Schott. Parallel searching in the plane. *Computational Geometry*, 5:143–154, 1995.

4 A. Beck. On the linear search problem. *Israel Journal of Mathematics*, 2(4):221–228, 1964.

5 A. Bonato and R. Nowakowski. *The Game of Cops and Robbers on Graphs*. American Mathematical Society, 2011.

6 S. Brandt, K.-T. Forster, B. Richner, and R. Wattenhofer. Wireless evacuation on m rays with k searchers. In *Proceedings of SIROCCO 2017, to appear*, 2017.

7 M. Chrobak, L. Gasieniec, T. Gorry, and R. Martin. Group search on the line. In *Proceedings of SOFSEM 2015: 41st International Conference on Current Trends in Theory and Practice of Computer Science*, pages 164–176, 2015.

8 J. Czyzowicz, L. Gasieniec, T. Gorry, E. Kranakis, R. Martin, and D. Pajak. Evacuating robots via unknown exit in a disk. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC 2014), Austin, TX, USA*, pages 122–136, 2014.

9 J. Czyzowicz, K. Georgiou, M. Godon, E. Kranakis, D. Krizanc, W. Rytter, and M. Wlodarczyk. Evacuation from a disc in the presence of a faulty robot. In *Proceedings of SIROCCO 2017, to appear*, 2017.

10 J. Czyzowicz, K. Georgiou, E. Kranakis, L. Narayanan, J. Opatrny, and B. Vogtenhuber. Evacuating robots from a disk using face-to-face communication (extended abstract). In *Proceedings of the 9th International Conference on Algorithms and Complexity (CIAC 2015), Paris, France*, pages 140–152, 2015.

11 J. Czyzowicz, E. Kranakis, D. Krizanc, L. Narayanan, J. Opatrny, and S. M. Shende. Wireless autonomous robot evacuation from equilateral triangles and squares. In *Proceedings of the 14th International Conference on Ad-hoc, Mobile, and Wireless Networks (ADHOC-NOW 2015), Athens, Greece*, pages 181–194, 2015.

12 Y. Emek, T. Langner, J. Uitto, and R. Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming, (ICALP 2014), Part II*, pages 471–482, 2014.

13 O. Feinerman and A. Korman. Theoretical distributed computing meets biology: A review. In *Proceedings of the 9th International Conference on Distributed Computing and Internet Technology, (ICDCIT 2013), Bhubaneswar, India*, pages 1–18, 2013.

14 O. Feinerman, A. Korman, Z. Lotker, and J.-S. Sereni. Collaborative search on the plane without communication. In *ACM Symposium on Principles of Distributed Computing (PODC 2012), Funchal, Madeira, Portugal*, pages 77–86, 2012.

15 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *Theor. Comput. Sci.*, 337(1-3):147–168, 2005.

16 S. K. Ghosh and R. Klein. Online algorithms for searching and exploration in the plane. *Computer Science Review*, 4(4):189–201, 2010.

17 L. Hua and E. K. P. Chong. Search on lines and graphs. In *Proceedings of the 48th IEEE Conference on Decision and Control, CDC 2009, China*, pages 5780–5785, 2009.

18 A. Jez and J. Lopuszanski. On the two-dimensional cow search problem. *Information Processing Letters*, 109(11):543–547, 2009.

19 C. Lenzen, N. A. Lynch, C. C. Newport, and T. Radeva. Trade-offs between selection complexity and performance when searching the plane without communication. In *ACM Symposium on Principles of Distributed Comp. (PODC 2014),*, pages 252–261, 2014.

20 A. López-Ortiz and G. Sweet. Parallel searching on a lattice. In *Proceedings of the 13th Canadian Conference on Computational Geometry (CCCG 2001)*, pages 125–128, 2001.

21 P. Nahin. *Chases and Escapes: The Mathematics of Pursuit and Evasion*. Princeton University Press, 2012.

22 L. D. Stone. *Theory of Optimal Search*. Academic Press New York, 1975.

23 I. Thompson. *Understanding Maple*. Cambridge University Press, 2016.

Model Checking of Robot Gathering

Ha Thi Thu Doan[1], François Bonnet[2], and Kazuhiro Ogata[3]

1 **Japan Advanced Institute of Science and Technology, Nomi, Japan**
doanha@jaist.ac.jp
2 **Graduate School of Engineering, Osaka University, Osaka, Japan**
francois@cy2sec.comm.eng.osaka-u.ac.jp
3 **Japan Advanced Institute of Science and Technology, Nomi, Japan**
ogata@jaist.ac.jp

Abstract

Recent advances in distributed computing highlight models and algorithms for autonomous mobile robots that self-organize and cooperate together in order to solve a global objective. As results, a large number of algorithms have been proposed. These algorithms are given together with proofs to assess their correctness. However, those proofs are informal, which are error prone. This paper presents our study on formal verification of mobile robot algorithms. We first propose a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions. We specify this formal model in Maude, a specification and programming language based on rewriting logic. We then use its model checker to formally verify an algorithm for robot gathering problem on ring enjoys some desired properties. As the result of the model checking, counterexamples have been found. We detect the sources of some unforeseen design errors. We, furthermore, give our interpretations of these errors.

1998 ACM Subject Classification D.2.4 Software/Program Verification, H.3.4 Systems and Software

Keywords and phrases Mobile Robot, Robot Gathering, Formal Verification, Model Checking, Maude

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.12

1 Introduction

Theoretical research on distributed mobile robot focuses mainly on computability aspects; the goal is to determine whether a problem can be solved given some assumptions, such as synchrony, multiplicity detection and chirality. Various models and problems have been proposed for the last two decades (*e.g.* Suzuki and Yamashita first paper [18], or the book from Flocchini *et al.* [15]). A few major models have emerged (*e.g.* ASYNC, weak multiplicity detection) for which some of the main problems (*e.g.* gathering, exploration) are now fully understood. In this work, we would like to take a step back and look at what have been accomplished so far. Now that results are stable, it is the right time to spend some energy carefully reviewing the proposed algorithms.

There have been already some efforts to unify and formalize existing results [9, 10, 8, 14]. While earlier papers described algorithms based on a (potentially long and cryptic) list of rules (*e.g.* [4]), more recent publications usually describe algorithms based on mathematical abstractions and real pseudo-codes (*e.g.* [8]). Naturally, at the same time, proofs become also more formal. We believe this is going, of course, in the good direction. Being closer to the mathematical world also makes these algorithms/proof well suited to be checked systematically.

Formal verification and related work. Classic distributed algorithms have been formally verified [19, 17, 13, 12]. We aim to obtain similar achievements for distributed robot algorithms. However, due to the mobility aspect, mobile robot algorithms are often complex, arguably even more complex than classic distributed systems. This inherent difficulty explains probably the limited number of attempts in obtaining formal verifications.

Recently Courtieu *et al.* [7] and Balabonski *et al.* [2] formally proved the correctness of two gathering algorithms using the Coq proof assistant. In both cases, robots move on the continuous 2-dimensional plane. The difference lies in the timing model; the first paper considers the semi-synchronous (SSYNC) model while the second studies the fully-synchronous (FSYNC) model (and remove some other assumptions). The asynchronous model (ASYNC) is not considered in these papers since the gathering problem is generally not solvable in ASYNC in continuous space (except for trivial cases).

Closely related to our current approach, Berard *et al.* [3] and Doan *et al.* [11] analyze the perpetual exploration algorithm described in [4]. This algorithm, while being quite simple (only 3 robots on the ring and 8 rules), was refuted. Both papers use similar techniques; they formally specify the algorithm and then encode in Linear Temporal Logic (LTL) the properties that should be satisfied. The first paper uses DiVinE and ITS tools, while the second one uses Maude.

This paper is somehow an extension of [11] to a different problem. Note however that this is not an easy generalization; our new formalization should accommodate any number of robots instead of only three. The previous studies [7, 2, 3, 11] basically consider either FSYNC or ASYNC without multiplicity assumption. The lack of multiplicity, of course, simplifies the model. Here we consider ASYNC and the possibility for multiple robots to be located on the same node.

Context. In this paper, we restrict our attention to discrete models only, and more specifically to the ring topology. About timing assumption, we consider the more general asynchronous model ASYNC. In addition, we take into account multiplicity assumption, which makes it much harder to formalize mobile robot algorithms. We present how to formalize a mobile robot algorithm as a state machine and then specify the state machine in Maude. Maude is a rewriting logic-based programming and specification language and equipped with a powerful system (or environment). Rewriting logic makes it possible to naturally specify dynamic systems, and the Maude system has an LTL model checker. We have demonstrated in [13, 12, 11] that Maude allows us to specify distributed algorithms/systems more succinctly than others. For instance, it supports *associative* and *commutative* operator attributes that are very necessary to concisely specify mobile robot algorithms as showing in [11].

We then use the Maude LTL model checker to formally verify an algorithm for robot gathering problem on ring enjoys desired properties. We focus on the gathering problem and analyze the algorithm proposed by D'Angelo *et al.* [8] as a case study. As the result of the model checking, counterexamples have been found. We detect the sources of unforeseen design errors. We, furthermore, give our explanations on these errors.

Contributions. The paper presents a study on how to specify and model check mobile robot algorithms. The contribution of this paper is the proof by example that formal methods must be used to verify distributed robot algorithms. Indeed, even algorithms described and proven using mathematical abstractions (may) still contain errors. While some of them are minor and could have been detected by a careful reader (simple typos), some errors would have been almost *impossible* to detect without model-checking. Said differently, we believe that

informal and semi-formal mathematical proofs are not enough for this kind of algorithms. The complexity of analyzing all situations may be too high for human brains.

Two main contributions are: (1) a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions – our model is general enough and could be applied to other problems (in the ring); (2) a refutation by model checking that the algorithm enjoys desired properties – in detail, the algorithm contains design errors that prevent robots from gathering into one location.

The additional contributions are a preliminary set of Maude modules that could be re-used for future verifications and the interpretations of the errors found. While it may not be straightforward to understand Maude formalism, we hope that it may still be useful for other people.

Outline. Section 2 describes the model, problem, and the main ideas of the algorithm under study. Section 3 presents how we formalize and then specify the system in Maude. Section 4 describes our model checking of the algorithm, showing and explaining the result. Section 5 finally concludes the paper.

2 Robots Gathering in the Ring under ASYNC

As mentioned in the Introduction, we consider the classic gathering problem in the ring under the asynchronous scheduler (ASYNC). We analyze the most general algorithm that solves the problem for (almost) all initial configurations. The considered algorithm is said to be general in a sense that it solves the problem for all valid (*i.e.* without multiplicity) initial configurations outside of $NG \cup SP4$, where (1) NG is the set of Non-Gatherable configurations (such as periodic configurations), from which it is impossible to gather robots, and (2) $SP4$ is the "small" set of SPecial configurations with 4 robots from which it is still unknown whether it is possible to gather robots (Bonnet *et al.* gave a partial answer [5]).

In the remainder of this section, we succinctly present the model, the problem, and the algorithm under study. For each part, a more complete description can be found in the original paper [8].

2.1 Computational Model

This model description is adapted from [5] for our specific context. The ring is *anonymous*, that is, there is neither node nor edge labeling. The robots are *identical*, *i.e.*, they are indistinguishable and all execute the same algorithm. Moreover, the robots are *oblivious* and *disoriented*, meaning that they have no memory of past actions, and they share no common orientation (no chirality).

The robots cannot explicitly communicate, but have the ability to sense their environment and see the relative positions of the other robots, in their local coordinate system. We assume the global weak multiplicity detection; each robot can distinguish whether a node is empty, occupied by one robot, or more than one robot. When there is strictly more than one robot, we use the term *multiplicity*. Robots follow a three-phase behavior: *Look*, *Compute*, and *Move*. During its Look phase a robot takes a snapshot of all robots' positions. The collected information (position of the other robots in the egocentric view) is used in the Compute phase during which the robot decides to move or stay idle. In the Move phase, the robot may move to one of the two adjacent nodes, as computed in the previous phase. The moves are assumed to be instantaneous which means that, during a Look phase, robots can be located on nodes only.

The computational model we consider is the classic asynchronous ASYNC model [15]. It means that, the start and duration of each Look-Compute phases and the start of each Move phase of each robot are arbitrary and determined by an adversary. Note that it is possible for a robot to make a move based on a previously observed configuration which is not the current one anymore (*e.g.* if its Look phase occurred before the Move phase of another robot).

A move that has been computed (during a Compute phase) but not yet executed (in the subsequent Move phase) is called a *pending move*.

2.2 Gathering Problem

The gathering problem requires each robot to terminate on the same node. The problem is solved if all robots are on the same location and there is no pending move.

2.3 Gathering Algorithm

This paper analyzes the algorithm [8] for robot gathering. The algorithm executes these four phases sequentially:

1. Starting from an initial configuration without multiplicity, the algorithm executes a procedure MULTIPLICITY-CREATION that creates either one or two symmetric multiplicities.
2. A second phase named COLLECT consists in moving all but four robots in the previously created multiplicities.
3. A third phase called MULTIPLICITY-CONVERGENCE makes the two multiplicities to merge into a single one.
4. Finally the phase CONVERGENCE allows the remaining single robots to join the unique multiplicity, which concludes the gathering.

This is a short (partially incorrect) summary. In some *rare* cases, the sequence of four phases may be temporarily broken. (*e.g.* the system may go back to the COLLECT phase while executing the CONVERGENCE phase). But eventually all robots should gather at the same location.

The algorithm contains some specific subroutines for configurations with four or six robots.[1] At the moment, we decided not to include them in our analysis; as in the paper, they could be dealt separately.

Plaintext vs. Pseudo-code. In [8], the algorithms are described and explained in plaintext and also given in term of pseudo-codes. We think that the pseudo-code version contains less ambiguities than the plaintext version. In (pseudo-)code, there is usually no place for interpretation; it is thus either correct or incorrect. Since our goal is to formally model-check, we believe that it makes more sense to base our analysis on the most formal available version. That is why this paper analyzes the pseudo-code version of the algorithms.

This is certainly an arguable decision. Indeed, some of the errors, at least the ones from Section 4.3, do not exist in the plaintext description of the algorithms. Other detected errors may or may not exist in the plaintext version. We can not conclude anything about the correctness of the plaintext algorithm since it is subject to interpretation.

[1] The precise algorithm is not given for four or six robots, but refers to other papers.

3 Formal Model for Mobile Robot Algorithms

In this section, we propose a formal model for mobile robot algorithms on anonymous ring shape network under multiplicity and asynchrony assumptions. Unavoidably, multiplicity and asynchrony make arduous to formalize the systems. We pay much attention to this problem and solve it in our model. To describe the model, we use state machines. A state machine consists of a set S of states, some of which are initial states, and a binary relation $T \subseteq S \times S$. Each element $(s, s') \in T$ is called a state transition from s to s'. We then transfer this model into Maude specification language. In Maude, the basic units of specifications and programs are modules. A module contains syntax declarations, providing suitable language to describe a system. There are two kinds of modules: functional modules and system modules. Functional modules are those in which data structures, such as pair and sequence, are specified, and system modules are those in which systems, such as distributed systems, are specified. A distributed system is formalized as a state machine and then the state machine is specified in Maude as a system module.

For these systems, a state of the system is called a configuration. A configuration is described in terms of a view starting from any robot and traversing the ring in one arbitrary direction. When a robot wakes up, it takes the snapshot of the current configuration of the system, and computes a move (called a computed move) based on this snapshot. The computed move is either staying idle or moving to one of its adjacent nodes. In the latter case, it moves to the adjacent node, eventually. In the following part, Maude notation is used to describe state machines. We consider how to express a state and how to describe an event as a state transition.

3.1 State Expressions

We denote a robot as a pair $\langle I, P \rangle$, where I denotes the size of the interval[2] between it and the next robot, and P denotes the computed move. The value of P could be nil, fc or $fc-$ (fc stands for $f$ollowing the $c$onfiguration). nil means that the robot has no pending move (*i.e* last computed move was idle). fc (resp. $fc-$) means that the robot has a pending move to the adjacent node located after it (resp. before it) following the direction of the configuration. Initially, the computed move of each robot is nil. If P is nil, the robot may be activated and will compute a new move and update P accordingly. If P is not nil, the robot may be activated and will execute the move and update P to nil after the move. We use the sort[3] *Pending* to denote computed moves and the sort *Pair* to denote pairs. They are expressed by the following operators that are constructors as specified with *ctor*.

op ⟨_, _⟩ : Int Pending → Pair [ctor] .

The sort *Int* is used for denoting integers. The operator ⟨_, _⟩ is used to construct *Pair*. For $c_1 \in Int$, $c_2 \in Pending$, $\langle c_1, c_2 \rangle \in Pair$.

A configuration is expressed as a sequence of pairs. It contains the information about the locations of all robots and their states. The corresponding sort is *Config*. Configurations are defined by the following operators.

[2] An interval is a maximal set of empty consecutive nodes.

[3] The types of data are called *sorts*. A sort denotes the set of elements in the same type. For example, the sort Nat is used to denote the set of Natural numbers. A sort is a subsort of another sort if and only if the set denoted by the former is a subset of the one by the latter, and the latter is called a supersort of the former.

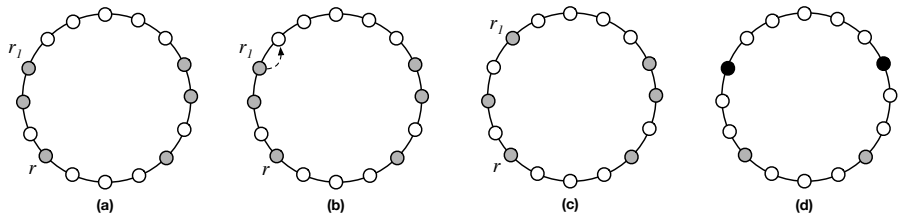


Figure 1 Some configurations. A dashed arrow represents a pending move. Black nodes represent multiplicities.

subsort Pair < Seq .
op empS : → Seq [ctor] .
op _ _ : Seq Seq → Seq [ctor assoc id: empS] .
op {_} : Seq → Config [ctor].

where the sort *Seq* is used for sequences of pairs. *empS* denotes the empty sequence of pairs. *Seq* is a supersort of *Pair*, which means that each *Pair* is treated as the singleton sequence only consisting of the pair. The juxtaposition operator _ _ is used to construct non-trivial sequences of pairs. For c_1, $c_2 \in$ *Seq*, c_1 $c_2 \in$ *Seq*. The juxtaposition operator _ _ is associative as specified with *assoc*, and *empS* is an identity of the operator specified with *id: empS*.

A configuration is of the form {_} of a sequence. States of the system are expressed as terms of the sort *Config*. A term of a sort S is a variable of S or $f(t_1, \ldots, t_n)$ if f is an operator declared as $f : S_1 \ldots S_n \to S$ $(n \geq 0)$ and $t_1, \ldots, t_n$ are terms of $S_1, \ldots, S_n$. If f has any underscores _, such as $\langle \_, \_ \rangle$, then a different notation than $f(t_1, \ldots, t_n)$ is used, such as $\langle I, P \rangle$ that is a term of the sort *Pair*, where I is term of the sort *Int* and P is a term of the sort *Pending*. Constructor terms are those consisting of constructors only and variables. Ground term are those having no variables. Ground constructor terms hence are those composed of constructors only and no variables . Ground constructor terms of the sort *Config* express concrete states of the system. For example, the initial configuration of the system as shown in Fig. 1(a) could be expressed as the view starting from the robot r in clockwise order, $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. Let us assume that robot r_1 is activated, takes a look at the configuration, and computes a move. Assuming that the move is to move to the node located after it, the system reaches the configuration as shown in Fig. 1(b). It is expressed as $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. If the robot r_1 is activated again, it executes the move; the system becomes as shown Fig. 1(c) which is expressed as $\{\langle 1, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$. A robot is not allowed to look at the second element of the pairs of other robots and it calculates a move based on its own view of the system. For example, the view of robot r_1 in Fig. 1(a) could be either $\{\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\}$ or $\{\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\}$ without knowing anything about clockwise order. It is worth noting that this allows us to guarantee that robots have no sense of direction and do not know the pending moves of other robots.

Due to the multiplicity assumption, it is possible that a robot moves to a node that is occupied by other robots. The node is, or becomes a multiplicity. There may be more than one of such robots in one multiplicity. Since the robots are anonymous, we can denote all of them by a pair $\langle I, P \rangle$ in which the value of I is set to the negative of the additional number of robots located on the multiplicity (-3 indicates 3 additional robots, which means a multiplicity of 4 robots). Note that this notation allows us to represent the exact number

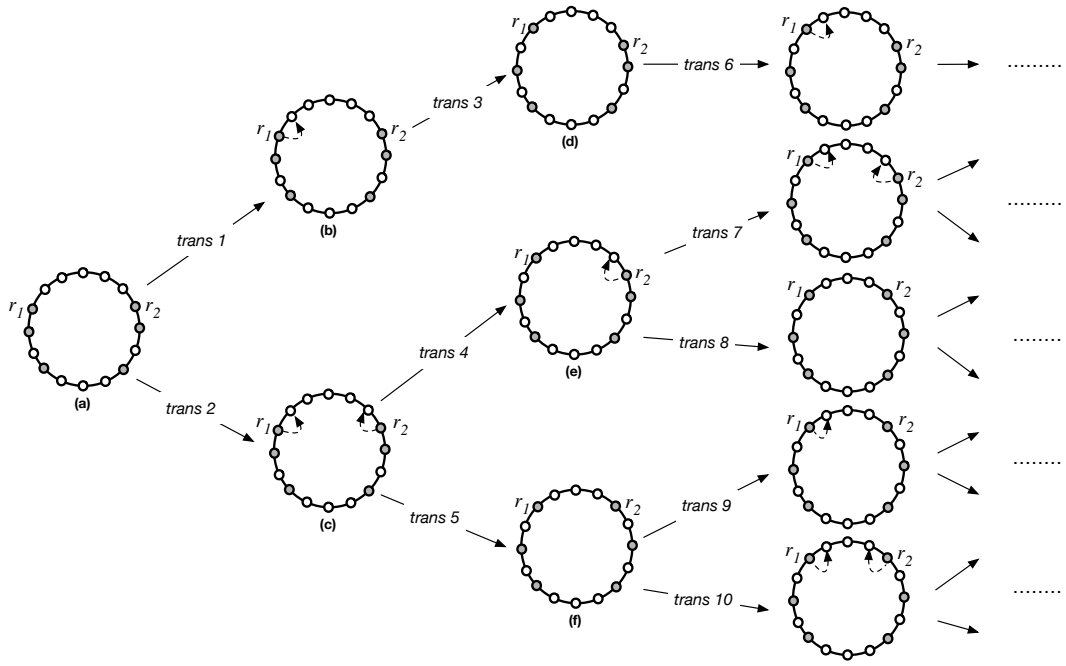


Figure 2 A transition graph of one specific initial configuration (a).

of robots in multiplicities. But robots do not have access to this information; they can only know if there is a multiplicity (*i.e.* a negative number). Our encoding allows a simple conversion to consider global strong multiplicity detection. For instance, the configuration as shown in Fig. 1(d) assuming that there are two robots in each multiplicity, is expressed as $\{\langle 2, nil\rangle\langle -1, nil\rangle\langle 5, nil\rangle\langle -1, nil\rangle\langle 2, nil\rangle\langle 3, nil\rangle\}$. We use this encoding to match as closely as possible the definitions introduced in [8].

3.2 State Transitions

Because the *Compute* phase uses the snapshot of the system taken in the *Look* phase as input and a robot does not perform any movements during two phases, to model the system, we combine the two phases into one called the *Look-Compute* phase in which a robot takes the snapshot of the system and computes a move. When either (1) a robot takes the snapshot of the system and then computes a move, or (2) a robot executes its pending move, the current configuration of the system changes to another. Such changes are called a state transition (or a transition). A transition is expressed as a pair (l, r), where l and r are configurations.

Let us examine the following scenario. Given an initial configuration as shown in Fig. 2(a) and assumed that both robots r_1 and r_2 are allowed to move, it may happen that only one robot (assuming r_1) looks at the system and computes a move, or both r_1 and r_2 do. In the former case, the configuration of the system is transferred to the one as shown in Fig. 2(b). The transition is named $trans1$ and expressed by the pair $(\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle, \langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle)$. In the latter case, the configuration of Fig. 2(c) is established. The configuration of Fig. 2(d) is obtained after r_1 in the configuration of Fig. 2(b) executes its pending move. The configuration of Fig. 2(e) and Fig. 2(f) are obtained from the configuration of Fig. 2(c). The graph in Fig. 2 shows possible transitions from the initial configuration. A sequence of transitions

starting from an initial configuration, e.g $trans1$, $trans3$, $trans6$, ..., is called a possible execution. There may exist more than one execution from a given initial configuration.

We describe the actions of robots as transition rules. A transition rule is described in the form of a rewrite rule. Each rewrite rule is defined only over *Config* that does not have any sub-sorts and in the form L ⇒ R such that L only consists of constructors and variables. We give here a simple example to explain how an action can be expressed as a transition rule. The following transition rule describes the action corresponding to a robot executing its pending move when there is no multiplicity in the system.

crl [fc-pending] : {S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2} ⇒ {S1 ⟨ I1 + 1, P ⟩ ⟨ I2 - 1, nil ⟩ S2}
if nonMul({S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2}).

where $S1$, $S2 \in Seq$, $I1$, $I2 \in Int$ and $P \in Pending$ are variables of those sorts; The function *nonMul* returns *true* when the configuration has no multiplicity and *false* otherwise.

The above rule is a conditional writing rule and the condition is specified in the if part. The rule then will be applied if the condition is satisfied. The configuration {S1 ⟨ I1, P ⟩ ⟨ I2, fc ⟩ S2} expresses any state such that the robot ⟨ I2, fc ⟩ holds a pending move fc and the robot before it is ⟨ I1, P ⟩. Such a state may have some more robots before and after the two robots that are expressed as $S1$ and $S2$, respectively. The ground constructor term $\{\langle 1, nil\rangle\langle 0, nil\rangle\langle 5, fc\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$ expresses the state as shown in Fig. 2(b). There is no multiplicity in this configuration. The left-hand side of the above rewrite rule *fc-pending* matches this ground term by substituting *S1, I1, P, I2* and *S2* with $\langle 1, nil\rangle$, 0, nil, 5 and $\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle$, and the rewrite rule can be applied to the term, changing it to $\{\langle 1, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 3, nil\rangle\}$ expressing the state as shown in Fig. 2(d). In this way, a rewrite rule expresses a set of state transitions.

To make T, a set of transitions (s, s'), from rewrite rules L ⇒ R, let σ be a substitution from the variables in L to appropriate constructor terms, $\sigma(L)$ is a constructor term, but $\sigma(R)$ is not necessarily, and then it is necessary to reduce $\sigma(R)$ with equations. Let $nf(t)$ be the term obtained by reducing t with equations. So, $(\sigma(L), nf(\sigma(R)))$ is a state transition obtained from L ⇒ R. Let $\sigma(L) \Rightarrow nf(\sigma(R))$ be called a ground instance of L ⇒ R.

▶ **Definition 1** (TR_{RS})**.** Let TR_{RS} be the set of all ground instances of the all transition rules.

3.3 Formal Model

We formalize a mobile robot system as a state machine. The state machine includes the set of all possible states as the set of all ground constructor terms S_{RS}, the set of initial states I_{RS} and the binary relation over states T_{RS}. I_{RS} is a subset of S_{RS} such that for each state $s \in S_{RS}$, there is no multiplicity and the configuration does not belong to $NG \cup SP4$. T_{RS} is the binary relation over S_{RS} made from TR_{RS}.

▶ **Definition 2** (M_{RS})**.** The state machine formalizing a mobile robot system is M_{RS}, where

1. S_{RS} is the set of all ground constructor terms whose sorts are *Config*;
2. I_{RS} is a subset of S_{RS} such that $(\forall s \in I_{RS})$ (numMul(s) = 0) and (not ng&sp4(s));
3. T_{RS} is the binary relation over S_{RS} defined as follows:
   {(l, r) | l ⇒ r $\in TR_{RS}$}.

The function numMul counts the number of the multiplicities in the system and the function ng&sp4 returns *true* when a configuration is in $NG \cup SP4$ and *false* otherwise.

We specify this formal model in Maude specification languages. Note that our specification is coherent [6]. The source files are available for download [1].

4 Model Checking the Algorithm

Model checking is a verification technique that explores all possible system states and checks whether a desired property that should be satisfied by an algorithm is satisfied. The desired property is required to be formally expressed. A model checker then verifies whether the formula is satisfied for all possible executions. If the formula is not satisfied, a counterexample is found. We specify the formal model of the system in Maude. We then apply Maude LTL model checker to formally verify the algorithm. The original paper [8] has given very important lemmas, such as *Lemma 5, 6, and 7*, that state properties that need to be satisfied at the end of each phase. These lemmas are used to model check the algorithm. We have formally expressed these lemmas as LTL formulas [16]. We give here the formalization of *Lemma 6* as an example. *Lemma 6* states a property that must be satisfied at the end of the COLLECT phase. Namely, it states that the configuration obtained at the end of the COLLECT phase contains two multiplicities and satisfies a condition (called located condition) that the configuration needs to be in some specific configurations in which robots are located in some specific locations. Note that the COLLECT phase can start only if the initial configuration is symmetric or at one step from specific symmetric configurations. To model check this lemma, we expressed it as an LTL formula. We define the atomic propositions *endOfColl* and *coll* as follows:

$C \models \text{endOfColl} = \text{checkOfColl}(C)$.

$C \models \text{coll} = \text{checkColl}(C)$.

$\text{checkColl}(C) = \text{checkAllowedSym}(C)$ and $(\text{numMul}(C) == 2)$ and $\text{checkCondition}(C)$.

The function *checkOfColl* checks whether a system state C is at the end of the COLLECT phase. The atomic proposition *endOfColl*, thus, is true if and only if the COLLECT phase phase has just finished. The function *checkAllowedSym* checks whether a configuration is an allowed symmetric configuration. The function *checkCondition* returns *true* when the configuration satisfies the located condition and *false* otherwise. The atomic proposition *coll*, thus, is *true* when the configuration is an allowed symmetric configuration containing two multiplicities[4] and satisfies the located condition and *false* otherwise.

The lemma then is formally expressed as an LTL formula as follows.

$\text{lemma6} = \Box\ (\text{endOfColl} \rightarrow \text{coll}) \wedge \Diamond\ \text{endOfColl}$.

where $\Box$ stands for always (globally) and $\Diamond$ stands for eventually (in the future).

Intuitively, the formula states that it is always true that the phase COLLECT will finally terminate and whenever the phase has been just over, then *coll* is true. This means that the *Lemma 6* is satisfied at the end of the COLLECT phase.

This formula is used to conduct the model checking for the algorithm. To model check, we separate the formula into two sub-formulas and model check them separately in order to easily detect the source of errors (if some are found). Namely, we use the two following formulas:

$\text{lemma6-1} = \Diamond\ \text{endOfColl}$.

$\text{lemma6-2} = \Box\ (\text{endOfColl} \rightarrow \text{coll})$.

As the result of the model checking, counterexamples are found. A counterexample is of the form of a possible execution: it includes all visited states and the sequence of transition rules applied. This helps to analyze counterexamples to detect the source of the detected errors. We present two counterexamples as follows:

[4] The mathematical notation == stands for equivalence.

The first one results from the model checking of the formula lemma6-1.

```
reduce in EXPERIMENT : modelCheck(init, lemma6-1) .
rewrites: 89524096 in 40088ms cpu (40173ms real) (2233163 rewrites/second)
result ModelCheckResult: counterexample(
      {{< 0,nil > < 1,nil > < 0,nil > < 3,nil>
        < 0,nil > < 1,nil > < 0,nil > < 0,nil >},'w5-fo}
      {{< 0,nil > < 1,nil > < 0,nil > < 3,nil > < 0,fc >
        < 1,nil > < 0,nil > < 0,nil >},'FC22-pending}
      {{< 0,nil > < 1,nil > < 0,nil > < 4,nil > < -1,nil >
        < 1,nil > < 0,nil > < 0,nil >},'coll-a-1-fo2}
      {{< 0,nil > < 1,nil > < 0,nil > < 4,fc- > < -1,nil >
        < 1,nil > < 0,nil > < 0,nil >},'FC-23-pending},
      {{< 0,nil > < 1,nil > < -1,nil > < 5,nil > < -1,nil >
        < 1,nil > < 0,nil > < 0,nil >},deadlock})
```

The counterexample shows that the system falls into a deadlock state. Indeed, the configuration $\{\langle 0, nil\rangle\langle 1, nil\rangle\langle -1, nil\rangle\langle 5, nil\rangle\langle -1, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 0, nil\rangle\}$ belongs to the kind of configurations that need to be handled by the COLLECT phase. However, the COLLECT phase could not deal with it. Thus, the system is not able to reach a valid state at the end of the COLLECT phase. Analyzing this counterexample, we detect the error, which is described later in Section 4.1. This also means that the algorithm fails in gathering all robots in the same location.

The second counter-example results from the model checking of the formula lemma6-2.

```
reduce in EXPERIMENT : modelCheck(init, lemma6-2) .
rewrites: 15982060 in 7064ms cpu (7114ms real) (2262435 rewrites/second)
result ModelCheckResult: counterexample(
    {{< 0,nil > < 1,nil > < 0,nil > < 3,nil >
      < 0,nil > < 1,nil > < 0,nil > < 0,nil >},'w5-fo}
    {{< 0,nil > < 1,nil > < 0,nil > < 3,nil >
      < 0,fc > < 1,nil > < 0,nil > < 0,nil >},'FC22-pending}
    {{< 0,nil > < 1,nil > < 0,nil > < 4,nil >
      < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'coll-a-1-fo1}
    {{< 0,nil > < 1,nil > < 0,fc- > < 4,nil >
      < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'FC-23-pending},
    {{< 0,nil > < 0,nil > < 1,nil > < 4,nil >
      < -1,nil > < 1,nil > < 0,nil > < 0,nil >},'ofColl})
```

This counterexample occurs because the state

$$\{\langle 0, nil\rangle\langle 0, nil\rangle\langle 1, nil\rangle\langle 4, nil\rangle\langle -1, nil\rangle\langle 1, nil\rangle\langle 0, nil\rangle\langle 0, nil\rangle\}$$

is at the end of the COLLECT phase, but the atomic propositions *coll* returns *false*. Specifically, at end of the COLLECT phase, the configuration does not contain two multiplicities and satisfy the located condition.

Since counterexamples are found, the lemma does not hold. The remainder of this section reports on some errors that we have found. In addition, we also give our opinions about the origins of these errors.

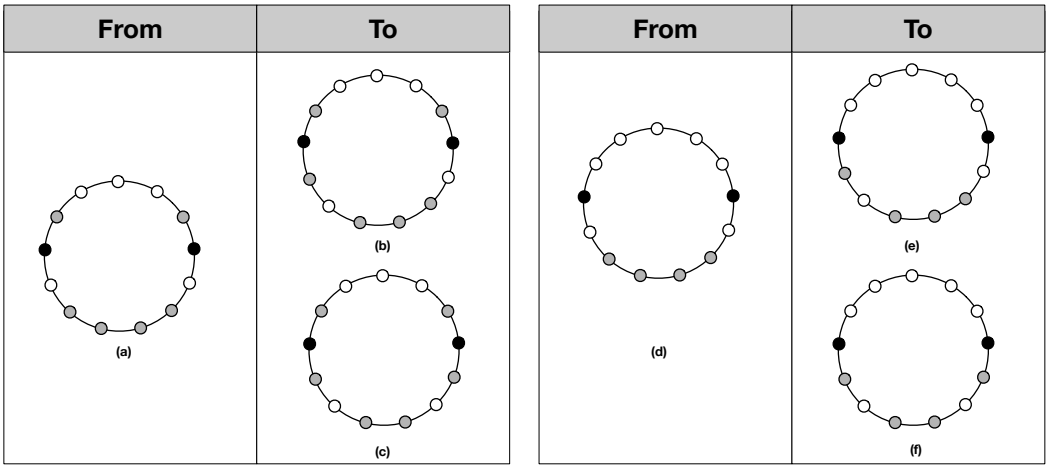


Figure 3 Expected executions for the two specific symmetric configurations with two multiplicities.

4.1 Omission of Special Cases

We report here the error that corresponds to the first counterexample. The error occurs when the algorithm deals with symmetric configurations with two multiplicities. Single robots in such configurations should move such that the configuration eventually reaches a symmetric one with (i) size nodes occupied, (ii) two multiplicities, (iii) two robots adjacent to these multiplicities, and (iv) other robots in specific locations. However, it is not straightforward to design a strategy for these robots. Two examples are shown in Fig. 3. For the configuration of Fig. 3(a), the two symmetric robots are expected to move such that either the configuration of Fig. 3(b) or Fig. 3(c) is obtained. In the same way, the configurations of Fig. 3(e) and Fig. 3(f) are obtained from the configuration of Fig. 3(d).

Unfortunately, a counterexample is found. Our model checking detects that the algorithm does not work correctly for the configuration of Fig. 3(d). The two single robots do not move as expected. The same configuration is obtained instead of the configurations of Fig. 3(e), 3(f). Indeed, the algorithm only works correctly for configurations in which there are two single robot adjacent to the two multiplicities, *e.g.* the configurations similar to the one of Fig. 3(a), but it does not work for configurations such as the one of Fig. 3(d).

This error exists because some cases were missing. For instance, the configurations as the configuration of Fig. 3(d) are not considered. That is why the algorithm is not able to deal with them. This would be very difficult to detect without the help of an automatic tool. The error leads to the fact that the system will fall to a deadlock state and all robots in the system are unable to locate at the same location. This error could be fixed by finding a strategy to deal with these case. However, we need to take care of the fact that the new strategies may be conflicting with exiting strategies.

4.2 Design Errors (difficult to detect by mathematical proof)

This section report errors of a different kind compared to the one of Section 4.1. One of the main issue to be handled by the algorithm concerns symmetric configurations in which two symmetric robots are supposed to move symmetrically. Let us explain the idea with an example. For the symmetric configuration as shown in Fig. 4(a), the two symmetric robots r and r_1 are allowed to symmetrically move. It may happen that both r and r_1 move and

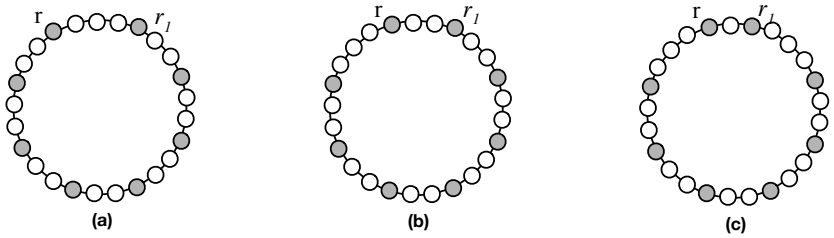


Figure 4 Three configurations where (a) is some initial configuration, (b) is the configuration obtained if only one robot moves, and (c) the one obtained if both symmetric robots move.

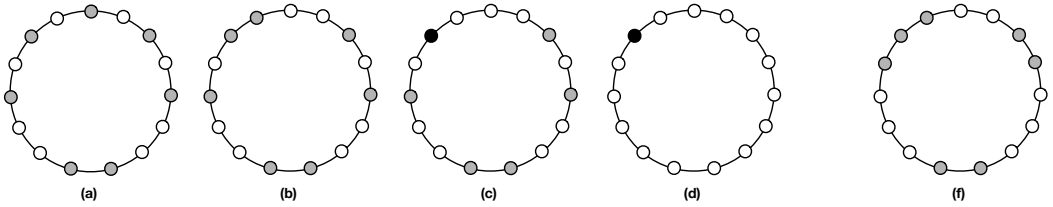


Figure 5 Five configurations. The gathering algorithm should follow the sequence (a), (b), (c), (d), but the configuration (f) is actually obtained from (b).

the configuration of Fig. 4(c) is obtained. It may also happen that only r moves, while r_1 already computed the move, but has not yet moved (means that it holds a pending move) or r_1 has not yet performed its Look-Compute phase and the configuration of Fig. 4(b) is obtained. This configuration is an asymmetric configuration that may contain a possible pending move (only robot r_1 knows if it has already computed its move; other robots do not know it). The procedures CHECK-REDUCTION and PENDING-REDUCTION are designed to deal with this case. The robots in such configuration are supposed to detect this situation and the robot r_1 is expected to move in order to reach the configuration of Fig. 4(c). It is not so difficult to design procedures that let robots in these configurations to recognized these configurations and move as expected. However, the problem is when these procedures are included in the entire program.

As written in [8], the procedure CHECK-REDUCTION also recognizes some other configurations where PENDING-REDUCTION should not be applied. One of them is the configuration depicted in Fig. 5(b). The authors use this configuration as an example to explain the algorithm (Figure 3 in [8]). The algorithm is expected to work as follows: since the configuration of Fig. 5(a) is symmetric with one robot on the axis, the robot on the axis has to move, obtaining the configuration of Fig. 5(b). This configuration is asymmetric and and contains only one *supermin*[5]. There is one important point emphasized by authors: the robots can recognize that there are no pending moves in this configuration. Therefore, the unique supermin is reduced until a multiplicity is created. In this example, the configuration Fig. 5(c) is obtained, and then all robots join the unique multiplicity one-by-one, until achieving the gathering as in Fig. 5(d). However, the algorithm actually works as follows: from configuration of Fig. 5(b), the robot which is supposed to create a multiplicity does not move. Instead two other robots move and we obtain the configuration of Fig. 5(f) (instead of Fig. 5(c)). Checking each step of the execution and the transition rules applied, we discover that the source of this error: Robots do not recognize the correct type

[5] The definition of *supermin* is given in [8]. One characteristic of a *supermin* is that it is smallest interval.

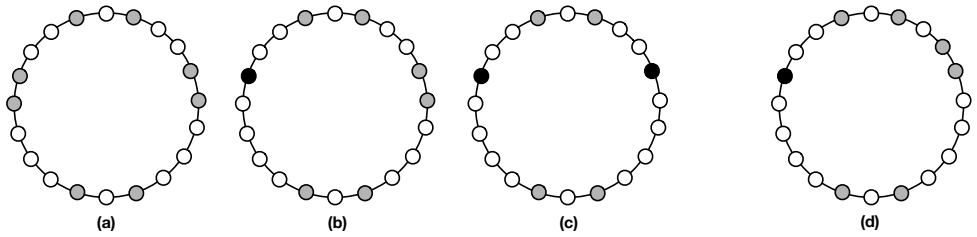


Figure 6 Four configurations. The gathering algorithm should follow the sequence (a), (b), (c), but the configuration (d) is actually obtained from (b).

of the configuration of Fig. 5(b), which is classified into a group named W7 that performs the procedure CHECK-REDUCTION and PENDING-REDUCTION. When executing the procedure CHECK-REDUCTION on this configuration, robots incorrectly categorize it as an as asymmetric configuration with possible pending moves. Then the PENDING-REDUCTION is executed while it should not be for this configuration. Thus, the two symmetric robots decide to move. The configuration of Fig. 5(f) is obtained. This means that the CHECK-REDUCTION and PENDING-REDUCTION procedures are not correct. This error would be very difficult to detect manually. Indeed, the procedures are correct for the configurations for which they are supposed to be used. It is only incorrect because it is also applied to configurations that are not supposed to use these procedures[6].

The second error is in the COLLECT phase. When entering into this phase, the configuration of the system belongs to one of three categories. One of them is called COLL-A-1. They are asymmetric configurations with one multiplicity that can be obtained from symmetric configurations and while satisfying also some other conditions. To simulate how the algorithm works, the authors give the scenario depicted in Fig. 6. The configuration of Fig. 6(b) is in COLL-A-1. It is at one move from the symmetric configuration of Fig. 6(a). When the configuration is in COLL-A-1, the algorithm checks whether the current configuration satisfies some conditions. If the conditions are satisfied, it moves the robot and re-establish the previous axis of symmetry, leading to a symmetric configuration with two multiplicities. In this specific case, the configuration of Fig. 6(c) is obtained.

Unfortunately, instead of moving the robot and re-establishes the previous axis of symmetry, the algorithm moves both adjacent robots in the same direction. That means the symmetric configuration with two multiplicities is not obtained. In this specific case, the configuration of Fig. 6(d) is obtained. This error is somehow similar to the previous design error, but at a different level; robot instead of configuration. Here, the robot that is supposed to move (from the plaintext description of the algorithm) really moves, but an additional robot also moves.

These design errors prevent the robots from gathering. Said differently, the algorithm fails in gathering all robots in one location. Both errors could certainly be fixed by including additional tests before computing the moves. However adding these tests may lead to other problems and is therefore not straightforward.

[6] As explained in Section 2, we analyze the pseudo-code of the algorithms [8]. It is unclear whether such error exists in the informal plaintext description of the algorithm.

4.3 Some minor errors

We also want to report some other small errors. They could be detected by carefully checking the pseudo-code. However, it may be difficult to find where they are in the code. Fortunately, we can also detect them by model checking.

4.3.1 Minor "typo" error

If a configuration is periodic, it is impossible to gather all robots to one location. Therefore, it is important to detect whether a configuration is periodic or not. To check the periodicity, the authors give the following procedure:

Input: a configuration $C = (q_0, q_1, ..., q_j)$.

Output: **true** if C is periodic, **false** otherwise.

1. *periodic* := **false** .
2. **for** $i := 0, 1, ..., j$ **do if** $C = C_i$ **then** *periodic* = **true**.
3. **return** *periodic* .

where *periodic* is a boolean variable. It is expected to return *true* if the configuration C is periodic, *false* otherwise. We discover that *periodic* returns *true* also for aperiodic configurations. This is obviously wrong. The source of the error is that the condition $C = C_0$ is true for any configuration C. Thus, the procedure always returns *true* for any input configuration C. One needs to simply start iterations from $i = 1$ instead of $i = 0$.

4.3.2 Error of inattention

The second error detected is in the main procedure for phase MULTIPLICITY-CREATION whose purpose is to create one multiplicity or two symmetric multiplicities. Since this is the first phase of the algorithm, which deals with initial configurations, all possible initial configurations are partitioned into seven groups. One of them is called *W6*. In this case, the procedure is given as follows:

Input: CT, $C = Q(r) = (q_0, q_1, ..., q_j)$.

Case $CT = W6$

1. $C' := (q_0 + 1, q_1 - 1, ..., q_j)$.
2. **if** $C' = \overline{C'}$ *and* q_0 *is odd* **then** move towards q_0;
3. **else**
   $C'' := (q_0, ..., q_{j-1} - 1, q_j + 1)$.
   if $C''_j = \overline{C''_j}$ *and* q_j *is odd* **then** move towards q_j;

where $C = Q(r) = (q_0, q_1, ..., q_j)$ is the configuration that is perceived by robot r. $\overline{C}$ corresponds to $(q_0, q_j, q_{j-1}, ...q_1)$ in the case where C = $(q_0, q_1, ..., q_{j-1}, q_j)$.

This code is supposed to handle asymmetric configurations that could have been obtained from a symmetric configurations with an odd number of nodes and a node-edge symmetric axis. In the above code, C' (or C'') is the configuration of the symmetric configuration. The intention of the authors is to move a robot r in order to reach a new symmetric configuration with the same original axis. One example taken from the Appendix of [8] is given in Fig. 7. The configuration of Fig. 7(b) can be obtained from the symmetric configuration of Fig. 7(a). Thus, the robot r is supposed to move and the configuration of Fig. 7(c) is obtained.

However, the algorithm does not make the robot r to move. We found that the error lies in the conditions: q_0 *(or* q_j*) is odd*. Let us take a look at symmetric configurations (e.g as shown in Fig. 7(a)) with odd number of nodes and one axis passing through an edge, we can see that the interval crossed by the axis is on an odd interval. That means that $q_0 + 1$ (or $q_j + 1$) is odd, but not q_0 (or q_j). The error can be fixed by updating the conditions to test if $q_0 + 1$ (or $q_j + 1$) is odd. This error was probably made due to the confusion between the configurations C and C' (or C'').

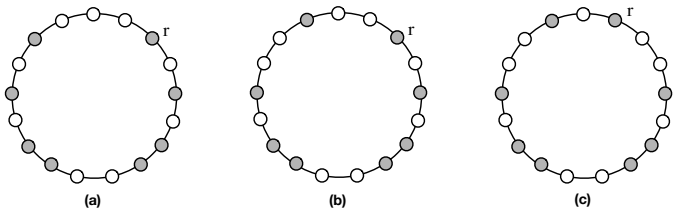


Figure 7 Three configurations, where (a) may lead to (b), and (c) presents the expected behavior of the algorithm for the specific asymmetric configuration (b).

4.4 Summary of model checking

It is very common to use case analysis techniques to tackle non-trivial problems, such as robot gathering. The authors in [8] have split the problem into many cases and used different strategies to deal with them. In detail, they have partitioned not only the initial configurations, but also the configurations in each phase. Since the authors need to consider a large number of cases, there is a risk of missing some cases. Moreover, since different strategies are used for each case, there could be some conflicts between these strategies. To prove the algorithm, the authors have to consider all possible cases. However, it is exhausting for a person to figure out all possible executions. Fortunately, a model checker strongly supports to automatically check all possible executions. We have showed how to formally express the desired properties as LTL formulas and conduct the model checking of the algorithm. The model checking has found counterexamples. Analyzing these counterexamples, we found the source of some errors that would be difficult to detect by handmade proof. These errors make the algorithm fail in gathering all robots in one location. We also gave some explanations about why such errors may have occurred. We hope it may be useful to avoid them in the future.

5 Conclusion

How to formalize and model check a new software system is always challenging. We show in this paper how to formalize and model check a new form of distributed system. We have demonstrated the usefulness of model checking techniques to formally verify mobile robot algorithms. Although informal proofs have been given in [8] to guarantee the correctness of the algorithm, there remain some mistakes that are subtle and not easy to find by carefully checking the algorithm, even by experts. We want to emphasize that our goal is not to blame the authors or reviewers of the paper. When reading the paper, we also missed most of these errors. But it means that it is indeed difficult and therefore we should really consider using formal methods to check such algorithms.

References

1 Our Maude source files. URL: `https://figshare.com/s/3c0a6fdb0e4fec3b0b08`.

2 T. Balabonski, A. Delga, L. Rieg, S. Tixeuil, and X. Urbain. Synchronous gathering without multiplicity detection: A certified algorithm. In *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 10083 of *LNCS*, pages 7–19. Springer, 2016.

3 B. Bérard, P. Lafourcade, L. Millet, M. Potop-Butucaru, Y. Thierry-Mieg, and S. Tixeuil. Formal verification of mobile robot protocols. *Distributed Computing*, 29(6):459–487, 2016.

4 L. Blin, A. Milani, M. Potop-Butucaru, and S. Tixeuil. Exclusive perpetual ring exploration without chirality. In *Proceedings of the 24th International Symposium on Distributed Computing*, volume 6343 of *LNCS*, pages 312–327. Springer, 2010.

5 F. Bonnet, M. Potop-Butucaru, and S. Tixeuil. Asynchronous gathering in rings with 4 robots. In *Proceedings of the 15th International Conference on Ad-hoc, Mobile, and Wireless Networks*, volume 9724 of *LNCS*, pages 311–324. Springer, 2016.

6 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

7 P Courtieu, L. Rieg, S Tixeuil, and X. Urbain. Certified universal gathering in r^2 for oblivious mobile robots. In *Proceedings of the 30th International Symposium on Distributed Computing*, volume 9888 of *LNCS*, pages 187–200. Springer, 2016.

8 G. D'Angelo, G. Di Stefano, and A. Navarra. Gathering on rings under the look–compute–move model. *Distributed Computing*, 27:255–285, March 2014.

9 G. D'Angelo, G. Di Stefano, A. Navarra, N. Nisse, and K. Suchan. Computing on rings by oblivious robots: A unified approach for different tasks. *Algorithmica*, 72(4):1055–1096, 2015.

10 G. D'Angelo, A. Navarra, and N. Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 28(1):17–48, 2017.

11 H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of a mobile robots perpetual exploration algorithm. In *Proceedings of the 6th International Workshop on Structured Object-Oriented Formal Language and Method (SOFL+MSVL 2016)*, volume 10189 of *LNCS*, pages 201–219. Springer, 2017.

12 H. T. T. Doan, F. Bonnet, and K. Ogata. Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In *Proceedings of The 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, pages 1586–1596, 2017.

13 H. T. T. Doan, W. Zhang, M. Zhang, and K. Ogata. Model checking chandy-lamport distributed snapshot algorithm revisited. In *Proceedings of the 2nd International Symposium on Dependable Computing and Internet of Things*, pages 7–19, 2015.

14 P. Flocchini, D. Ilcinkas, A. Pelc, and N. Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

15 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool Publishers, 2012.

16 Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

17 I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.

18 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

19 T. Tsuchiya and A. Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5):341–358, 2011.

Plane Formation by Synchronous Mobile Robots without Chirality*†

Yusaku Tomita[1], Yukiko Yamauchi[2], Shuji Kijima[3], and Masafumi Yamashita[4]

1 **Kyushu University, Fukuoka, Japan.**
`tomita@tcslab.csce.kyushu-u.ac.jp`
2 **Kyushu University, Fukuoka, Japan.**
`yamauchi@inf.kyushu-u.ac.jp`
3 **Kyushu University, Fukuoka, Japan.**
`kijima@inf.kyushu-u.ac.jp`
4 **Kyushu University, Fukuoka, Japan.**
`mak@inf.kyushu-u.ac.jp`

Abstract

We consider a distributed system consisting of autonomous mobile computing entities called *robots* moving in the three-dimensional space (3D-space). The robots are *anonymous*, *oblivious*, *fully-synchronous* and have neither any access to the global coordinate system nor any explicit communication medium. Each robot cooperates with other robots by observing the positions of other robots in its *local coordinate system*. One of the most fundamental agreement problems in 3D-space is the *plane formation problem* that requires the robots to land on a common plane, that is not predefined. This problem is not always solvable because of the impossibility of symmetry breaking. While existing results assume that the robots agree on the handedness of their local coordinate systems, we remove the assumption and consider the robots *without chirality*. The robots without chirality can never break the symmetry consisting of *rotation symmetry* and *reflection symmetry*. Such symmetry in 3D-space is fully described by 17 symmetry types each of which forms a group. We extend the notion of *symmetricity* [Suzuki and Yamashita, SIAM J. Compt. 1999] [Yamauchi et al., PODC 2016] to cover these 17 symmetry groups. Then we give a characterization of initial configurations from which the fully-synchronous robots without chirality can form a plane in terms of symmetricity.

1998 ACM Subject Classification I.2.11 Distributed Artificial Intelligence

Keywords and phrases Autonomous mobile robots, plane formation problem, symmetry breaking, group theory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.13

1 Introduction

Self-organization of autonomous mobile computing entities, called agents, robots, or particles, has gained much attention as real hardware robots, such as wheeled robots and drones, become cheaper and widely available. Applications in robotics cover various cooperative behavior, such as surveillance, exploration, and transportation. A swarm of autonomous

* This work was partially supported by a Grant-in-Aid for Scientific Research on Innovative Areas "Molecular Robotics" (No.24104003 and No.15H00821) of MEXT and MEXT/JSPS KAKENHI Grant Numbers JP15K15938, JP17K19982, JP15H02666, and JP15K11987.
† A full version of the paper is available at `https://arxiv.org/abs/1705.06521`

mobile computing entities has also gained much attention in design and analysis of chemical reactions of molecules, cells, and DNA strands. Cooperative behavior in biological systems can be considered as self-organization of mobile computing entities, such as foraging ants, schooling of fishes, and migration of birds. In theoretical distributed computing, a number of models for these swarm systems have been proposed. For example, the *autonomous mobile robot model* considers an autonomous mobile robot with very weak capabilities, i.e., without any global positioning system nor any communication medium [15]. The *amoebot model* is inspired by amoeba [7], and the *population protocol model* considers delay-tolerant networks [1]. The *sticky particle model* considers particles (or square tiles) moving to the same direction according to an external force, like magnetic force [3]. Self-organization has been formalized as the *formation problem* that requires the entities to form a given shape (robot systems [15, 17, 12, 19], amoebot systems [8], and sticky particles [14]). In this paper, we consider the formation problem by autonomous mobile robots in the three-dimensional Euclidean space (3D-space). We focus on the robots with weakest abilities and investigate an essential element that determines their self-organization ability.

We adopt the conventional autonomous mobile robot model [10]. A *robot system* consists of a set of autonomous mobile robots, each of which is an *anonymous* (indistinguishable) point in a specified space. Each robot has neither any access to the global coordinate system nor any explicit communication medium. The robots cooperate with each other by observing the positions of other robots, but their local observations may be inconsistent because each robot uses its own local coordinate system. Formally, each robot repeats a *Look-Compute-Move cycle*, where it takes a snapshot (i.e., local observation) of other robots in the Look phase, computes its next position with a common algorithm in the Compute phase, and moves to the next position in the Move phase. Each local coordinate system is an orthogonal coordinate system in the specified space. It has an arbitrary unit length and the directions of coordinate axes are arbitrary. Further more, they might not agree on handedness, i.e., they can be right-handed or left-handed, thus they lack *chirality*. A robot is *oblivious* if in a Compute phase, it does not remember the past observations and the past computations and can use the observation obtained in the Look phase of the current cycle. Otherwise, a robot is *non-oblivious*, which means it is equipped with persistent local memory. A movement is *rigid* if each robot reaches its next position in each move phase, otherwise *non-rigid*. A *configuration* of a robot system is the set of positions of the robots observed in the global coordinate system, in other words, a set of points. There are three timing models for the execution of cycles: In the *fully-synchronous (FSYNC) model*, the robots execute the i-th Look-Compute-Move cycle at the same time. In the *semi-synchronous (SSYNC) model*, executions of cycles are synchronized but not all robots execute the i-th cycle at the same time. In the *asynchronous (ASYNC) model*, no assumption is made except that the length of each cycle is finite.

The *pattern formation problem* requires the robots to form a given target pattern. A sequence of results by Suzuki and Yamashita [15, 17] and Fujinaga et al. [12] showed that the set of formable pattern by anonymous robots is determined by initial *symmetry* among the robots regardless of asynchrony and obliviousness. Consider an initial configuration of four oblivious FSYNC robots with rigid movement, where they form a square and their local coordinate systems are symmetric regarding the center (Figure 1a). The four robots cannot break their symmetry forever from this initial configuration because they have the same local observation and they execute a common algorithm synchronously. Thus the robots cannot form, for example, a line. This worst case occurs in the ASYNC model (thus in the SSYNC model) with non-rigid movement because they allow stronger synchronization and rigid movement.

(a) Rotation regarding the center. **(b)** Reflection regarding a mirror plane.

Figure 1 Symmetric initial positions and local coordinate systems in 2D-space. A solid arrow shows an x-axis and a broken arrow shows an y-axis.

On the other hand, there are several crucial elements that reduce the set of formable patterns, thus the self-organization power of a robot system. Dieudonné et al. showed that anonymity prevents the robots from forming an arbitrary target pattern [9]. They showed that unique leader enables the oblivious ASYNC (thus SSYNC and FSYNC) robots to form an arbitrary target pattern. Yamauchi and Yamashita showed that when the *visibility* (observation range) is limited, the oblivious FSYNC (thus SSYNC and ASYNC) robots may increase their symmetry that cannot be resolved later [20]. Thus, limited visibility substantially reduces the formable patterns. Cicerone et al. showed that an important building block of the pattern formation algorithm for oblivious ASYNC robots in [12] does not work correctly without chirality [4]. They pointed out that robots without chirality may forever move symmetrically regarding the *center of rotation* or an *axis of reflection* (Figure 1b). Interestingly, the common cause of these three crucial elements is also *symmetry* among the robots.

As shown in Figure 1, the symmetry that the robots can never break is caused by symmetric local coordinate systems. When the robots have chirality, a local coordinate system is obtained by a uniform scaling, a translation, a rotation, or a combination of them on the global coordinate system. Thus we can obtain symmetric local coordinate systems of the robots with chirality by symmetric rotation operations. When the robots lack chirality, reflection by mirror planes are added to symmetric rotation operations because reflection changes handedness of local coordinate systems. For the robots with chirality in 2D-space, Yamashita et al. introduced the notion of *symmetricity* of a set of points. We consider a decomposition of a set P of points into regular m-gons centered at one point. We consider that one point is a regular 1-gon with an arbitrary center and two points form a regular 2-gon with the center being the midpoint. The maximum value of such m is the symmetricity $\rho(P)$ of P in 2D-space. When $\rho(P)$ is greater than one, the common center is the center of the smallest enclosing circle of P, denoted by $c(P)$, and $\rho(P)$ is generally the order of the cyclic group that acts on P. This definition is based on the fact that for each of these regular m-gons there exist symmetric local coordinate systems regarding $c(P)$, and the robots with such initial local coordinate systems cannot break the regular m-gon forever. However, when $c(P) \in P$, we have $\rho(P) = 1$, meaning that the symmetry of P can be broken. This is achieved by the robot on $c(P)$ leaving its current position.

Yamauchi et al. extended symmetricity to 3D-space [19]. Symmetric rotation operations in 3D-space form $SO(3)$ of infinite order, and its subgroups with finite order are the *cyclic groups*, the *dihedral groups*, the *tetrahedral group*, the *octahedral group*, and the *icosahedral group*. Each rotation group is recognized as the set of symmetric rotation operations on a prism, a pyramid, a regular tetrahedron, a regular octahedron, and a regular icosahedron,

respectively. The *symmetricity* $\varrho(P)$ of a set P of points is the set of rotation groups G such that the group action of G on P divides P into $|G|$-sets where $|G|$ is the order of G. They also showed that the robots on rotation axes can eliminate the rotation axes by leaving their current position in the same way as 2D-space. This symmetry breaking results in, for example, the fact that the robots can form a regular octahedron from a cube.

However, existing definitions of symmetricity assume chirality. In this paper, we consider symmetry among robots without chirality in 3D-space, thus composite symmetry of rotation and reflection. The combination results in $O(3)$, and its subgroups with finite order consist of seventeen *symmetry groups*, that are fully studied in group theory [2, 6]. We extend the notion of symmetricity in [19] to these seventeen symmetry groups. Then we consider the *plane formation problem* that requires the robots in 3D-space to land on a common plane without making multiplicity. As already shown, the plane formation problem is not always solvable because of the impossibility of symmetry breaking [18]. We validate our definition of symmetricity by presenting a characterization of initial configurations from which the robots can solve the plane formation problem. We first show a necessary condition based on the fact that the robots cannot break their symmetricity forever. Then we show a matching sufficient condition with a plane formation algorithm because existing plane formation algorithms [18, 19] do not work without chirality. The highlight of our results is the fact that when an initial configuration contains an "empty" mirror plane that does not contain any robot, the robots cannot break their symmetry regarding this mirror plane, and they cannot form a plane. This results in a decrease of solvable instances (initial configurations) compared to the robots with chirality, i.e., when the robots with chirality initially occupy the vertices of a cube, they can form a plane, while the robots without chirality cannot. On the other hand, we show that when a mirror plane contains some robots, these robots can eliminate the mirror plane by leaving their current positions. Thus our results partly extend symmetry breaking phenomena of rotation axes to mirror planes.

By considering the robots with weakest observation abilities, we give the complete description of symmetry of the robots in 3D-space. On the other hand, we assume strong synchronization model and movement model, i.e., the FSYNC robots with rigid movement, while existing literature considers the plane formation problem for FSYNC robots with rigid movement [18] and for SSYNC robots with non-rigid movement [16]. The reason is twofold. First, in terms of symmetry, the worst case is caused by FSYNC rigid movement. Since such a worst case occurs in the SSYNC (thus ASYNC) model with non-rigid movement, the impossibility is essentially determined by FSYNC model with rigid movement. Second, although SSYNC (thus ASYNC) robots with non-rigid movement require carefully designed algorithms, we can extend our plane formation algorithm to the SSYNC robots with non-rigid movement in the same way as Uehara et al.'s techniques [16].

Related work

In terms of the pattern formation ability of a robot system, symmetry dominates other elements such as anonymity, obliviousness, visibility, and chirality. The set of formable patterns are characterized by the initial symmetricity among the robots. Regarding 2D-space, it has been shown that irrespective of obliviousness and asynchrony, the robots with chirality can form a target pattern F from an initial configuration P if and only if $\rho(P)$ divides $\rho(F)$ except the case where F is a point with multiplicity two [12, 15, 17]. The exception is called the *rendezvous problem*, which is trivially solvable by FSYNC robots while not solvable by SSYNC (thus ASYNC) robots. Yamauchi et al. partially generalized the results to 3D-space by showing that irrespective of obliviousness, the FSYNC robots with chirality can form a target pattern F from an initial configuration P if and only if $\varrho(P)$ is a subset of $\varrho(F)$ [19].

In terms of symmetricity in 2D-space, the point formation problem and the circle formation problem are expected to be solvable from any initial configuration. While the point formation of two SSYNC (thus ASYNC) robots is unsolvable, more than two ASYNC robots can always form a point [5]. Additionally, any number of oblivious ASYNC robots can form a circle without chirality [11, 13].

Fujinaga et al. investigated the *embedded pattern formation problem*, where the target pattern is given as a set of landmarks on the plane [12]. They showed that the oblivious ASYNC robots can form any embedded target pattern by showing an algorithm that is based on the "clockwise" minimum-weight perfect matching between the positions of the robots and the landmarks. Based on this clockwise matching algorithm, Fujinaga et al. presented a pattern formation algorithm for oblivious ASYNC robots with chirality [12]. Later Cicerone et al. pointed out that the clockwise matching algorithm does not work when the robots lack chirality, and showed a new embedded target pattern formation algorithm [4].

The plane formation problem was first introduced by Yamauchi et al. for FSYNC robots with chirality and rigid movement [18]. Part of the motivation was to use these existing distributed coordination techniques for 2D-space. However, they pointed that even when the robots with chirality finish plane formation, they may not agree on the clockwise direction (thus handedness) on the plane. Consider a configuration of two robots with right-handed x-y-z local coordinate systems on a plane, where the z-axis of one robot points to the upward and that of the other robot points to the downward. Then there is no way to make the two robots agree on the clockwise direction. This fact highlights the importance of distributed algorithms that do not require chirality.

2 Preliminary

2.1 Robot Model

Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of n anonymous robots, each of which is a point in 3D-space. We use r_i just for description. We consider discrete time $t = 0, 1, 2, \ldots$ and let $p_i(t) = (x_i(t), y_i(t), z_i(t)) \in \mathbb{R}^3$ be the position of r_i at time t in the global x-y-z coordinate system Z_0, where $\mathbb{R}$ is the set of real numbers. The *configuration* of R at time t is $P(t) = \{p_1(t), p_2(t), \ldots, p_n(t)\}$. We denote the set of all possible configurations of R by $\mathcal{P}_n^3$. We assume that the initial positions of robots are distinct, i.e., $p_i(0) \neq p_j(0)$ for $r_i \neq r_j$ and $|P(0)| = n$.[1] We also assume that $n \geq 4$ since any three robots are on one plane.

Each robot r_i has no access to the global coordinate system, and it uses its local x-y-z coordinate system Z_i. The origin of Z_i is the current position of r_i while the unit distance, the directions, and the orientations of the x, y, and z axes of Z_i are arbitrary and never change. It is appropriate to denote $Z_i(t)$, but we use a shorter description. Each Z_i is either right-handed or left-handed. Thus the robots do not have *chirality*. We denote the coordinates of a point p in Z_i by $Z_i(p)$.

We consider the *fully-synchronous (FSYNC) model*, where the robots start the t-th *Look-Compute-Move cycle* at the beginning of time $(t-1)$ and finishes it before time t $(t = 1, 2, \ldots)$. Each of the Look phase, the Compute phase, and the Move phase of a cycle is completely synchronized in each time step. At time t, each robot r_i obtains a set

[1] This assumption is necessary because when more than one robots are initially at one point, it is impossible to separate them by a deterministic algorithm. While the impossibility results in this paper considers executions possibly with multiplicities, the proposed algorithm does not make any multiplicity in any execution.

$Z_i(P(t)) = \{Z_i(p_1(t)), Z_i(p_2(t)), \ldots, Z_i(p_n(t))\}$ in the Look phase. We call $Z_i(P(t))$ the local observation of r_i at time t. Then r_i computes its next position by a common algorithm ψ in the Compute phase. A robot is *oblivious* if it does not remember the past observations and the past computations, thus the input to ψ is $Z_i(P(t))$. Otherwise, it is *non-oblivious* and the input to ψ contains the past observations and the past computations. Finally, r_i moves to the next point in the Move phase. We assume that each robot always reaches its next position in a move phase and we do not care for the route to reach there. Thus we consider *rigid movement*.

An execution of an algorithm ψ from an initial configuration $P(0)$ is a sequence of configurations $P(0), P(1), P(2), \ldots$. Given an algorithm ψ, there are multiple FSYNC executions starting from $P(0)$, however when the initial local coordinate systems of $P(0)$ and initial local memory content (if any) are fixed, the FSYNC execution is uniquely determined.

The *plane formation problem* requires that the robots land on a plane, which is not predefined, without making any multiplicity. Hence point formation is not a solution for the plane formation problem. We say that an algorithm ψ *forms a plane* from an initial configuration $P(0)$, if, regardless of the choice of initial local coordinate systems Z_i for each $r_i \in R$, in any execution $P(0), P(1), P(2) \ldots$ there exists a finite $t \geq 0$ such that (i) $P(t)$ is contained in a plane, (ii) $|P(t)| = n$, i.e., all robots occupy distinct positions, and (iii) once the system reaches $P(t)$, the robots do not move anymore.

For a set P of points, we denote the smallest enclosing ball (SEB) of P by $B(P)$ and its center by $b(P)$. A point on the sphere of a ball is said to be *on* the ball, and we assume that the *interior* and the *exterior* of a ball do not include its sphere. The *innermost empty ball* $I(P)$ is the ball whose center is $b(P)$, that contains no point of P in its interior and contains at least one point of P on its sphere. When all points of P are on $B(P)$, we say P is *spherical*. When the robots occupy the vertices of a polyhedron, we say that the robots *form* the polyhedron.

2.2 Symmetry by Rotations and Reflections

A symmetric rotation operation is an operation that rotates an object without changing its appearance. If we consider a unit ball, there are infinitely many symmetric rotation operations, and these operations form $SO(3)$. The subgroups of $SO(3)$ with finite order are the *cyclic groups* C_k $(k = 1, 2, \ldots)$[2], the *dihedral groups* D_ℓ $(\ell = 2, 3, \ldots)$, the *tetrahedral group* T, the *octahedral group* O, and the *icosahedral group* I. Each of them can be recognized as a set of rotation operations on a pyramid with a regular k-gon base, a prism with a regular ℓ-gon bases, a regular tetrahedron, a regular octahedron, and a regular icosahedron, respectively. In other words, they are determined by a set of rotation axes and their arrangements. A k-fold axis admits rotations by $2\pi i/k$ $(i = 1, 2, \ldots, k)$. These k operations form the *cyclic group* C_k of order k.

The *dihedral group* D_ℓ consists of a single ℓ-fold axis called the *principal axis* and ℓ 2-fold axes perpendicular to the principal axis, and its order is 2ℓ. We abuse the term "principal axis" for the single rotation axis of a cyclic group.

The *tetrahedral group* T consists of three 2-fold axes and four 3-fold axes, and its order is 12. The *octahedral group* O consists of six 2-fold axes, four 3-fold axes, and three 4-fold axes, and its order is 24. The *icosahedral group* I consists of fifteen 2-fold axes, ten 3-fold axes, and six 5-fold axes, and its order is 60. Clearly, multiple rotation axes of a rotation group

[2] C_1 consists of an identity element.

intersect at one point. We call the cyclic groups and the dihedral groups *2D rotation groups*, and we call the remaining three rotation groups T, O, and I *3D rotation groups* because a 3D rotation group does not act on a point on a plane.

Reflection by a mirror plane is an indirect symmetric operation. We consider composite symmetry of rotation axes and mirror planes, i.e., subgroups of $O(3)$ with finite order. Each symmetry type also forms a group. The *bilateral symmetry* C_s consists of one mirror plane and its order is 2. When there are more than one mirror planes, an intersection of mirror planes introduces a rotation axis. Clearly, the rotation axes and mirror planes of the symmetry type intersect at one point. The composition of C_k $(k > 1)$ and a horizontal mirror plane regarding the principal axis is denoted by C_{kh}. The order of C_{kh} is $2k$. The composition of C_k $(k > 1)$ and k vertical mirror planes each of which contains the principal axis is denoted by C_{kv}. The order of C_{kv} is $2k$.

The composition of D_ℓ $(\ell \geq 2)$ and a horizontal mirror plane regarding the principal axis is denoted by $D_{\ell h}$. However, this horizontal mirror plane together with rotation axes forces vertical mirror planes and the order of $D_{\ell h}$ is 4ℓ. The composition of D_ℓ $(\ell \geq 2)$ and ℓ vertical mirror planes is denoted by $D_{\ell v}$. The order of $D_{\ell v}$ is 4ℓ.

The composition of T and three mutually perpendicular mirror planes, each of which contains two 2-fold axes is denoted by T_h. The order of T_h is 24. The composition of T and six mirror planes, each of which contains two 3-fold axes is denoted by T_d. The order of T_d is 24. The composition of O and nine mirror planes is denoted by O_h. The order of O_h is 48. The composition of I and fifteen mirror planes, each of which contains two 5-fold axes is denoted by I_h. The order of I_h is 120.

Another type of composite symmetry is *rotation reflection* S_m, where a rotation regarding a single m-fold axis and reflection by a horizontal mirror plane are alternated. S_2 corresponds to the central inversion, which is denoted by C_i.

Let $\mathbb{S} = \{C_1, C_i, C_s, C_k, C_{kh}, C_{kv}, D_\ell, D_{\ell h}, D_{\ell v}, S_m, T, T_d, T_h, O, O_h, I, I_h \mid k = 2, 3, \ldots, \ell = 2, 3, \ldots, m = 3, 4, \ldots\}$. We call each element of $\mathbb{S}$ *symmetry group*. These seventeen types of symmetry groups describe all symmetry in 3D-space [6].

We denote the order of $G \in \mathbb{S}$ with $|G|$. When G' is a subgroup of G $(G, G' \in \mathbb{S})$, we denote it by $G' \preceq G$. If G' is a proper subgroup of G (i.e., $G \neq G'$), we denote it by $G' \prec G$. For example, we have $D_2 \prec T$, $T \prec O, I$, but $O \not\preceq I$. If $G \in \mathbb{S}$ has a k-fold axis, $C_{k'} \preceq G$ if k' divides k. For symmetry groups containing mirror planes, $T \prec T_h \prec O_h$ but $T_h \not\prec O$. For S_m, we have $C_{m/2} \prec S_m \prec C_{mh}$.

See Appendix A for the detailed description of symmetry groups.

2.3 Rotation Group, Symmetry Group, and Symmetricity

We start with the symmetry of positions of robots, called *symmetry group* of a configuration, that the robots can agree irrespective of their local coordinate systems. Then we introduce symmetry of local coordinate systems, called *symmetricity*, that the robots can never break. Note that the robots do not know local coordinate systems of other robots.

The *rotation group* $\gamma(P)$ of a set $P \in \mathcal{P}_n^3$ of points is the rotation group that acts on P and none of its proper supergroup in $\{C_k, D_\ell, T, O, I \mid k = 1, 2, \ldots, \ell = 2, 3, \ldots\}$ acts on P. The *symmetry group* $\theta(P)$ of P is the symmetry group that acts on P and none of its proper supergroup in $\mathbb{S}$ acts on P. Clearly, $\gamma(P)$ and $\theta(P)$ are uniquely determined,[3] and

[3] See for example [6], that shows an algorithm to uniquely determine the symmetry group of a polyhedra. The algorithm checks rotation axes, mirror planes, and a point of inversion. Since we consider a set of

Table 1 Rotation group, symmetry group, and symmetricity of regular polyhedra.

Polyhedron	Rotation group	Symmetry group	Symmetricity
Regular tetrahedron	T	T_d	$\{D_2, S_4\}$
Regular octahedron	O	O_h	$\{D_3, S_6\}$
Cube	O	O_h	$\{D_4, D_{2h}, D_{2v}, C_{4h}, S_4\}$
Regular dodecahedron	I	I_h	$\{D_5, D_2, S_{10}\}$
Regular icosahedron	I	I_h	$\{T, D_3, S_6\}$

$\gamma(P)$ is a subgroup of $\theta(P)$ ($\gamma(P) \preceq \theta(P)$). By the definition, when $\theta(P)$ is either C_1, C_i, C_s, $\gamma(P) = C_1$. Table 1 shows the rotation group of a set of vertices of each regular polyhedron.

The group action of $\theta(P)$ decomposes P into disjoint subsets. Let $Orb(p) = \{g * p \mid g \in \theta(P)\}$ be the orbit of $p \in P$ where $*$ denotes the action of g on s, and the orbit space $\{Orb(p) \mid p \in P\} = \{P_1, P_2, \ldots, P_m\}$ is called the $\theta(P)$-*decomposition* of P. Each element P_i is *transitive* because it is one orbit regarding $\theta(P)$.

Yamauchi et al. [18] showed that in configuration P without any multiplicity, the robots with chirality can agree on the $\gamma(P)$-decomposition $\{P_1, P_2, \ldots, P_m\}$ of P and a total ordering among the elements so that (i) P_1 is on $I(P)$, (ii) P_m is on $B(P)$, and (iii) P_{i+1} is not in the interior of the ball centered at $b(P)$ and containing P_i on its sphere. Though their techniques rely on chirality, we can extend it to robots without chirality. In [18], each robot translates its local observations to a "celestial map" by considering $I(P)$ as the earth. It considers that its current position is on the half line from $b(P)$ containing the north pole. Then, the robot selects an appropriate robot to define the prime meridian and translates the position of each robot to a triple consisting of its altitude, latitude, and longitude. The ordered sequence of these triples is the *local view* of the robots. However, the lack of chirality does not allow the robots to agree on the direction of longitude. Then we make a robot consider both directions and select the direction that produces the smallest sequence. In the same way as [18], we have the following property. We omit the proof because of the page limitation.

▶ **Lemma 1.** *Let $P \in \mathcal{P}^3_n$ and $\{P_1, P_2, \ldots, P_m\}$ be a configuration of n robots represented as a set of points and its $\theta(P)$-decomposition, respectively. Then we have the following two properties:*

1. *For each P_i $(i = 1, 2, \ldots, m)$, all robots in P_i have the same local view.*
2. *Any two robots, one in P_i and the other in P_j, have different local views, for all $i \neq j$.*

By Lemma 1, the robots can agree on a total ordering of the elements $P_1, P_2, \ldots, P_m$ by using the lexicographical ordering of the local views of each element. In the following, we assume that $P_1, P_2, \ldots, P_m$ is sorted by this ordering.

We denote the set of local coordinate systems for configuration P with a set Q of quadruples such that $Q = \{(o_i, x_i, y_i, z_i) \mid p_i \in P\}$ where o_i is the position of $p_i \in P$ (i.e., the origin of Z_i) and x_i, y_i, and z_i are the coordinates of $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ of Z_i observed in the global coordinate system Z_0. We use (P, Q) to explicitly show the set of local coordinate systems for P though Q contains P as $\{o_1, o_2, \ldots, o_n\}$. We define the *symmetry group* $\sigma(P, Q)$ of (P, Q) as the symmetry group that acts on (P, Q) and none of its proper supergroup in $\mathbb{S}$ acts on it. Clearly, we have $\sigma(P, Q) \preceq \theta(P)$. We define the $\sigma(P, Q)$-decomposition of (P, Q) in the same way as the $\theta(P)$-decomposition of P. We note that the robots of P cannot obtain Q nor $\sigma(P, Q)$ because they can observe only the positions of themselves.

points and their convex-hulls, we can use the same algorithm.

Given a set P of points, $\theta(P)$ determines the arrangement of its rotation axes and mirror planes in P. We thus use $\theta(P)$ and the arrangement of its rotation axes and mirror planes in P interchangeably. For two groups $G, H \in \mathbb{S}$, an *embedding* of G to H is an embedding of each rotation axis and each mirror plane of G to one of the rotation axes and one of the mirror planes of H with keeping their arrangement in G. Any k-fold axis of G is embedded so that it overlaps a k'-fold axis of H, where k divides k'. Observe that we can embed G to H if and only if $G \preceq H$.

We also consider a G-decomposition of a set P of points for some $G \prec \theta(P)$ $(G \in \mathbb{S})$ for an embedding of G in $\theta(P)$. Note that the robots cannot agree the ordering among the elements of such a G-decomposition of P.

▶ **Definition 2.** Let $P \in \mathcal{P}_n^3$ be a set of points. The symmetricity $\varrho(P)$ of P is the set of symmetry groups $G \in \mathbb{S}$ that acts on P (thus $G \preceq \theta(P)$) and there exists an embedding of G to $\theta(P)$ such that each element of the G-decomposition of P is a $|G|$-set.

We define $\varrho(P)$ as a set because the "maximal" symmetry group that satisfies the definition is not uniquely determined. Maximality means that there is no proper supergroup in $\mathbb{S}$ that satisfies the condition of Definition 2. When it is clear from the context, we denote $\varrho(P)$ by the set of such maximal elements. See Table 1 as an example.

We can rephrase the definition of symmetricity of P as a set of symmetry groups formed by rotation axes and mirror planes of $\theta(P)$ that do not contain any point of P. This is because a point on a rotation axis (a mirror plane, respectively) does not allow any decomposition into $|G|$-sets for any G that contains the rotation axis. [4]

We conclude this section with the following two lemmas, that validate the definition of symmetricity.

▶ **Lemma 3.** *For an arbitrary initial configuration $P \in \mathcal{P}_n^3$ and any $G \in \varrho(P)$, there exists a set of local coordinate systems Q such that $\sigma(P, Q) = G$.*

Proof. Let $\{P_1, P_2, \ldots, P_m\}$ be the G-decomposition of P for some embedding of G to $\theta(P)$. We show a construction of Q for P and G. Clearly, such embedding exists since $G \preceq \theta(P)$. From the definition, $|P_j| = |G|$ for $j = 1, 2, \ldots, m$. For each P_j, we arbitrary fix a local coordinate system of one robot $p_i \in P_j$. Then for each $p_k \in P_j$, there exists a unique element of G such that $p_k = g_k * p_i$ and $g_k \neq g_\ell$ if $p_k \neq p_\ell$ for any $p_\ell \in P_j$. Then we fix the local coordinate system of p_k by applying g_k to the local coordinate system of p_i. The local coordinate systems Q obtained by this procedure satisfies the property. ◀

Lemma 3 shows that there exists an arrangement of local coordinate systems Q for any initial configuration P and $G \in \varrho(P)$ such that $\sigma(P, Q) = G$. Then, Lemma 4 shows that the robots may forever caught in this initial symmetry.

▶ **Lemma 4.** *Irrespective of obliviousness, for an arbitrary initial configuration $P \in \mathcal{P}_n^3$, any $G \in \varrho(P)$, and any algorithm ψ, there exists an execution $P(0)(= P), P(1), P(2), \ldots$ such that $\theta(P(t)) \succeq G$.*

Proof. Let Q be initial local coordinate systems for P such that $\sigma(Q, P) = G$ for arbitrary $G \in \varrho(P)$. By Lemma 3, such Q always exists. Let $\{P_1, P_2, \ldots, P_m\}$ be the G-decomposition of P. From this arrangement of initial local coordinate systems, for each P_j $(j = 1, 2, \ldots, m)$, the robots forming P_j keep their symmetry group G forever for any algorithm ψ. We first show

[4] We assume that a set P of points does not contain any multiplicity. In other words, we consider an initial configuration P.

an induction for the oblivious FSYNC robots. For any $p_i, p_k \in P_j$, when $\psi(Z_i[P(0)]) = x$ holds, we have $\psi(Z_k[P(0)]) = x$ and $Z_0[Z_k(\psi(Z_k[P(0)]))] = g_k * Z_0[Z_i(\psi(Z_i[P(0)]))]$. Let $P_j(1) \subseteq P(1)$ be the positions of robots of P_j in $P(1)$. Thus $\theta(P_j(1)) = G$ and $\theta(P(1)) \succeq G$. By an easy induction for $t = 1, 2, \ldots$, we have the property for any $P(t)$.

Non-obliviousness does not improve the situation. When the initial memory contents are identical (for example, empty), the above discussion holds for the transition from $P(0)$ to $P(1)$. During this transition, the robots in the same element P_j obtain the same local observation, performs the same computation, and exhibits the same movement. Thus, their local memory content are still identical in $P(1)$ and they continue symmetric movement during the transition from $P(1)$ to $P(2)$. ◀

3 Impossibility of Plane Formation

The following theorem shows a necessary condition for the FSYNC robots without chirality to form a plane, that will be shown to be a sufficient condition in Section 4.

▶ **Theorem 5.** *Irrespective of obliviousness, the FSYNC robots without chirality can form a plane from an initial configuration P only if $\varrho(P)$ contains neither any 3D rotation group nor any 2D rotation group with horizontal mirror plane regarding the principal axis (except C_{2h} and S_m).*

Proof. Let ψ be an arbitrary plane formation algorithm for an initial configuration P such that $\varrho(P)$ contains a 3D rotation group, C_{kh} ($k \geq 3$), or $D_{\ell h}$ ($\ell \geq 2$). We have the following three cases.

Case A: $\varrho(P)$ contains C_{kh} for some $k \geq 3$.

Let Q be a set of initial local coordinate systems for P such that $\sigma(P, Q) = C_{kh} \in \varrho(P)$ ($k \geq 3$). From Lemma 4, irrespective of obliviousness, for any algorithm ψ, there exists an execution $P = P(0), P(1), P(2), \ldots$ such that $\theta(P(t)) \succeq C_{kh}$ for any $t \geq 0$. Assume that $P(t')$ be a terminal configuration. Then $\theta(P(t'))$ is a supergroup of C_{kh}, and $\theta(P(t'))$ has the mirror plane of $\theta(P)$. The robots are on this mirror plane, otherwise the robots are not on one plane because of their symmetry.

Let $\{P_1, P_2, \ldots, P_m\}$ be the $\sigma(P, Q)$-decomposition of $P(= P(0))$. For each P_i ($1 \leq i \leq m$) and $p \in P_i$, there exists $q \in P_i$ such that p and q are at symmetric positions regarding the mirror plane of C_{kh}. By Lemma 4, the robots of P_i move with keeping the rotation axis and the mirror plane of the embedding of C_{kh} in P. Thus p and q occupy the same point on the mirror plane of C_{kh} in $P(t')$. Hence the robots cannot avoid multiplicity and $P(t')$ is not a terminal configuration of the plane formation problem.

Case B: $\varrho(P)$ contains $D_{\ell h}$ for some $\ell \geq 2$.

Let Q be a set of initial local coordinate systems for P such that $\sigma(P, Q) = D_{\ell h} \in \varrho(P)$ ($\ell \geq 2$). By Lemma 4, irrespective of obliviousness, for any algorithm ψ, there exists an execution $P = P(0), P(1), P(2), \ldots$ such that $\theta(P(t)) \succeq D_{\ell h}$ for any $t \geq 0$. We have the same discussion as Case A. If there exists a terminal configuration, the robots are on the initial horizontal mirror plane of $D_{\ell h}$. Hence, the robots cannot avoid multiplicity and $P(t')$ is not a terminal configuration of the plane formation problem.

Case C: $\varrho(P)$ contains a 3D-rotation group.

The impossibility for this case has been shown for robots with chirality in [18] and the result holds for our robots because our model allows the robots with chirality. We note that when $\varrho(P)$ contains T_d, T_h, O_h or I_h, then it contains the corresponding rotation group because it is a subgroup without any mirror plane.

Consequently, when $\varrho(P)$ contains the above three types of symmetry groups, they cannot form a plane. ◀

4 Plane Formation Algorithm

In this section, we show a plane formation algorithm for oblivious FSYNC robots without chirality and prove our main theorem. Theorem 6 gives a necessary and sufficient condition for the FSYNC robots without chirality to solve the plane formation problem. In other words, it gives a characterization of the initial configurations from which the plane formation problem is solvable.

▶ **Theorem 6.** *Irrespective of obliviousness, the FSYNC robots without chirality can form a plane from an initial configuration P if and only if $\varrho(P)$ contains neither any 3D rotation group nor any 2D rotation group with horizontal mirror plane regarding the principal axis (except C_{2h} and S_m).*

The necessity is clear from Theorem 5. We prove the sufficiency by presenting a plane formation algorithm for solvable instances (i.e., initial configurations). Due to the page limitation, we show a sketch of the proposed algorithm.

By the condition of the theorem, solvable instances are classified into the following three types.

Type 1: Initial configuration P with $\theta(P) \in \{T, T_d, T_h, O, O_h, I, I_h\}$ is in this type. By the condition of Theorem 6, any initial configuration P of this type contains one of the following polyhedra as an element of its $\theta(P)$-decomposition, because some robots are on some rotation axes: a regular tetrahedron, a regular octahedron, a regular dodecahedron, and an icosidodecahedron.[5]

Type 2: Initial configuration P with $\theta(P) \in \{C_k, C_{kh}, C_{kv}, D_\ell, D_{\ell h}, D_{\ell v}, S_m | k = 2, 3, \ldots, \ell = 2, 3, \ldots, m = 3, 4, \ldots\}$ is in this type. By Theorem 6, $\theta(P)$ does not have the horizontal mirror plane or there are some robots on the horizontal mirror plane.

Type 3: Initial configuration P with $\theta(P) \in \{C_1, C_i, C_s\}$ is in this type.

The proposed algorithm handles these three types separately. The robots can agree on the type of the current configuration and they execute the corresponding algorithm. In a Type 1 initial configuration, the robots first break their symmetry and translates an initial Type 1 configuration to another Type 2 or Type 3 configuration. In a Type 2 initial configuration, the robots agree on a plane perpendicular to the principal axis and containing the center of their smallest enclosing ball, and land on the plane. In a Type 3 initial configuration, if it is an asymmetric configuration, the robots agree on a plane by using the total ordering among themselves (Lemma 1). Otherwise, the robots agree on a plane other than the mirror plane by using two elements of their $\theta(P)$-decomposition and land on it.

In the following, we use a point and a robot at the point interchangeably.

We put several preparation steps before the execution of the main algorithm. These steps are realized very easily in the FSYNC model without changing the initial symmetry of the robots.

Removing the center. When $b(P) \in P$, the robot on $b(P)$ leaves its current position so that a resulting configuration will be asymmetric.

For symmetry breaking of Type 1. If initial configuration P is Type 1, we send P_i of the $\theta(P)$-decomposition of P forming one of the specified polyhedra to the interior of $I(P)$.

[5] Points on rotation axes of a 3D rotation group also form a cube, a cuboctahedron, and a regular icosahedron. However, a cube allows D_{2h} to join its symmetricity, and the remaining two polyhedra allow $T(\prec O, I)$ to join their symmetricity.

We select P_i with the minimum index among the elements satisfying the condition to guarantee that no robot is on the track.

For symmetry breaking of Type 2. If initial configuration P is Type 2, we move P_j of the $\theta(P)$-decomposition of P on the horizontal mirror plane of $\theta(P)$ to the interior of $I(P)$. We select P_j with the minimum index among the elements satisfying the condition.

In the following, since the proposed algorithm is designed for the oblivious FSYNC robots, it is always described for a current configuration.

4.1 Symmetry Breaking

We consider a Type 1 configuration P. Let $\{P_1, P_2, \ldots, P_m\}$ be the the $\theta(P)$-decomposition of P. The preparation step guarantees that P_1 forms one of the following four polyhedra; a regular tetrahedron, a regular octahedron, a regular dodecahedron, and an icosidodecahedron. Then the proposed plane formation algorithm first makes the robots execute the *go-to-center algorithm* (Algorithm 1) proposed in [18]. Each robot of P_1 selects an adjacent face of the polyhedron and moves to the center of the selected face. But it stops ϵ before the center to avoid collisions. Clearly, Algorithm 1 does not depend on chirality. We have the following lemma for any execution of Algorithm 1.

▶ **Lemma 7.** *Let P and $\{P_1, P_2, \ldots, P_m\}$ be a Type 1 initial configuration and its $\theta(P)$-decomposition. Then there exists one element of P_i ($1 \leq i \leq m$) that forms one of the following polyhedra; a regular tetrahedron, a regular octahedron, a regular dodecahedron, or an icosidodecahedron. Then one step execution of Algorithm 1 translates P into another configuration P' that satisfies (i) $\gamma(P')$ is a 2D rotation group, and (ii) if $\gamma(P') \neq C_1, C_2$, $\theta(P')$ does not have any horizontal mirror plane.*

We present a sketch of the proof because of the page limitation. In [18], it is proved that Algorithm 1 breaks the rotation symmetry of the specified polyhedra and that the rotation group of any resulting configuration is a 2D rotation group. Since our robots lack chirality, we consider the combination of 2D rotation groups and mirror planes for a resulting configuration. We can show that the movement of the robots does not create any new mirror plane, and any resulting configuration does not have any horizontal mirror plane (except C_{2h}).

Since this symmetry breaking occurs in P_1, the (rotation) symmetry of the whole robots decreases to a 2D rotation group.

4.2 Landing Algorithm

In this section, we show a plane formation algorithm for Type 2 and Type 3 initial configurations. When $\gamma(P)$ of a current configuration P is a cyclic group or a dihedral group, our basic strategy is to make the robots agree on the plane perpendicular to the principal axis and containing $b(P)$. Each robot moves along a perpendicular to the agreed plane (Figure 2a). However, this simple strategy results in collisions on the agreed plane because the plane can be a mirror plane of $\theta(P)$. To avoid multiplicities, the proposed algorithm consumes five phases. The first three phases break the mirror plane of $\theta(P)$ and resolves the collisions on the target plane. The fourth phase makes the robots agree on the target plane and in the fifth phase each robot computes the destinations of all robots to avoid any collision.

In any configuration P, the robots execute the algorithm with the smallest phase number. The robots can easily agree on which phase to execute because the condition of the phases divide the set of all configurations with 2D rotation groups into disjoint subsets. Depending on the execution, some phases may be skipped.

Algorithm 1 Go-to-center algorithm for robot r_i [18].

Notation
P: Current configuration observed in Z_i.
$p_i \in P$: The position of r_i (i.e., the origin of Z_i).
$\{P_1, P_2, \ldots, P_m\}$: $\theta(P)$-decomposition of P, where P_1 forms one of the four polyhedra.
$\epsilon = \ell/100$, where ℓ is the length of an edge of the polyhedron that P_1 forms.
Algorithm
If $p_i \in P_1$ **then**
If P_1 is an icosidodecahedron **then**
Select an adjacent regular pentagon face of P_1.
Destination d is the point ϵ before the center of the face on the line from p_i to the center.
Else
// P_1 is a regular tetrahedron, a regular octahedron, or a regular dodecahedron.
Select an adjacent face of P_1.
Destination d is the point ϵ before the center of the face on the line from p_i to the center.
Endif
Move to d.
Endif

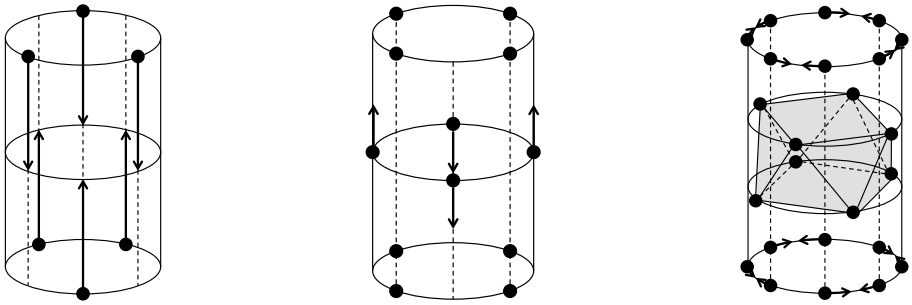
4.2.1 First phase: Removing the mirror plane

When $\theta(P)$ has a horizontal mirror plane, our basic strategy allows multiplicities. By the condition of Theorem 6, there is at least one element of the $\theta(P)$-decomposition of P on the horizontal mirror plane. To remove this mirror plane, we first make the robots of such an element leave their current positions (Fig. 2b). Intuitively, the first phase makes these robots select the upward direction or the downward direction regarding the horizontal mirror plane and they move to the selected directions. Any resulting configuration does not have the horizontal mirror plane any more because for each new positions of the robots, there is no corresponding point regarding the horizontal mirror plane.

In the following, we assume that when the current configuration P has a rotation axis, it does not have any horizontal mirror plane for the principal axis or $\theta(P) = C_{2h}$.

4.2.2 Second phase: Collision avoidance for dihedral groups

When $\gamma(P)$ is a dihedral group, say D_ℓ or $D_{\ell v}$, and the $\theta(P)$-decomposition of P contains a prism, our basic strategy makes multiplicities (Fig. 2c). By Theorem 6, $\theta(P)$ does not have a horizontal mirror plane, and there exists at least one element P_j that does not form such a prism. The robots of P_i circulates toward the nearest point of P_j and twist their prism. This movement resolves the collisions among the perpendiculars from the robots of P_i to the target plane.



(a) Antiprism **(b)** Horizontal mirror plane **(c)** Prism

Figure 2 Basic idea of the landing algorithm and collision avoidance. (a) Basic idea. (b) Removing the horizontal mirror plane. (c) Avoiding collisions by using an anti-prism.

4.2.3 Third phase: Collision avoidance on the principal axis

When $\gamma(P)$ is a dihedral group and some robots are on the principal axis, we need another trick to resolve the collisions of these robots. Clearly, these robots form element(s) of the $\theta(P)$-decomposition of P and the size of each of such elements P_k is two.

We also use an element of P_j that forms a "twisted" prism in the same way as the previous case. Each point of $p \in P_k$ selects the nearest point of P_j, however in this case, the robots of P_k do not move. Other robots consider the vertices of the twisted prism as possible destinations of this fictitious move.

4.2.4 Fourth phase: Agreement of the target plane

The robots agree on the target plane. Depending on $\theta(P)$ of the current configuration P, we have the following five cases:

(i) When $\gamma(P)$ is a cyclic group or a dihedral group, the robots agree on the plane perpendicular to the principal axis and containing $b(P)$. The exceptional case is when $\theta(P) = C_{2h}$. In this case, since the robots are not on one plane, there exists at least one element P_i of the $\theta(P)$-decomposition of P such that $|P_i| = 4$. Let i be the minimum index among such elements. Then, the robots agree on the plane formed by P_i. P_i forms a rectangle perpendicular to the horizontal mirror plane and the agreed plane is not a mirror plane for P.

(ii) When $\theta(P)$ is a rotation-reflection, the robots agree on the mirror plane of $\theta(P)$.

(iii) When $\theta(P)$ is a bilateral symmetry, the size of each element of the $\theta(P)$-decomposition of P is one or two. Since the robots are not on one plane, there exists at least one element P_i such that $|P_i| = 2$ and forms a perpendicular line regarding the mirror plane. If there is just one such element P_i, the robots agree on the plane defined by P_i (a line) and P_1 (a point). If $i = 1$, then the algorithm uses P_2 instead of P_1. Otherwise, let P_i and P_j be the elements with the minimum and second-minimum index of such elements. Then the robots agree on the plane defined by these two elements. The selected plane is not a mirror plane for P.

(iv) When $\theta(P)$ is a central inversion, the size of each element of the $\theta(P)$-decomposition of P is one or two. Since the robots are not on one plane, there are more than one elements of size two. Each of these elements form a line and they all intersect at the center of inversion. Let P_i and P_j be such elements with the minimum and second-minimum index. Then the robots agree on the plane defined by these two elements.

(v) When $\theta(P)$ is C_1, by Lemma 1, the robots can agree on the total ordering of themselves and agree on the plane defined by P_1, P_2, and P_3.

4.2.5 Fifth phase: Computation of final positions

Let P and $\{P_1, P_2, \ldots, P_m\}$ be the current configuration and its $\theta(P)$-decomposition. Through the previous four phases, for each element P_i $(i = 1, 2, \ldots, m)$, the foots of perpendiculars from the points of P_i to the target plane do not overlap. However, the foot of perpendiculars of different elements of the $\theta(P)$-decomposition of P may overlap.

Then all robots locally compute the landing positions of all robots in the order of $P_1, P_2, \ldots, P_m$. An element with a smaller index has a higher priority, i.e., if the landing positions of P_i are computed, any P_j $(j > i)$ avoids them with a common rule. For example, a point p is a destination of some robot in P_i, a small circle containing no other destination is drawn, and a destination of $q \in P_j$ is selected from this circle. Then this circle is considered as possible landing positions for P_j so that any collision will be avoided in the succeeding computation. Irrespective of handedness, all robots agree on all possible landing positions, and each robot computes its collision-free landing point. Finally, the robots move to their destinations in the same cycle.

As the non-oblivious robots can execute the proposed algorithm, we have the following theorem, that together with Theorem 5, proves our main theorem.

▶ **Theorem 8.** *Irrespective of obliviousness, the FSYNC robots without chirality can form a plane from an initial configuration P if $\varrho(P)$ contains neither any 3D rotation group nor any 2D rotation group with horizontal mirror plane regarding the principal axis (except C_{2h} and S_m).*

5 Conclusion

We considered the plane formation problem by FSYNC robots without chirality. We extended the notion of symmetricity in [19] to the composition of rotation symmetry and reflection symmetry. By using the symmetricity, we gave a characterization of initial configurations from which the FSYNC robots without chirality can form a plane. We also showed a plane formation algorithm for oblivious FSYNC robots without chirality.

One of the most important future directions is to investigate the effect of other system elements in 3D-space. The ASYNC model and the SSYNC model are not thoroughly discussed in 3D-space except the plane formation for SSYNC robots [16]. The effect of local visibility also remains to be investigated.

References

1 Dana Angluin, James Aspnes, Zoe Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing (PODC 2004)*, pages 290–299, 2004. doi:10.1145/1011767.1011810.

2 Mark A. Armstrong. *Groups and symmetry*. Springer-Verlag New York Inc., 1988.

3 Aaron Becker, Erik D. Demaine, Sándor P. Fekete, Golnaz Habibi, and James McLurkin. Reconfiguring massive particle swarms with limited, global control. In *Proceedings of the 9th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS 2013)*, pages 51–66, 2013. doi:10.1007/978-3-642-45346-5_5.

4 Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Asynchronous embedded pattern formation without orientation. In *Proceedings of the 30th International SymposiumDistributed on Computing (DISC 2016)*, pages 85–98, 2016. doi:10.1007/978-3-662-53426-7_7.

5 Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by mobile robots: gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012. doi:10.1137/100796534.

6 Peter R. Cromwell. *Polyhedra*. University Press, 1997.

7 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot – a new model for programmable matter. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 220–222, 2014. doi:10.1145/2612669.2612712.

8 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2016)*, pages 289–299, 2016. doi:10.1145/2935764.2935784.

9 Yoann Dieudonné, Franck Petit, and Vincent Villain. Leader election problem versus pattern formation problem. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC 2010)*, pages 267–281, 2010. doi:10.1007/978-3-642-15763-9_26.

10 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed computing by oblivious mobile robots*. Morgan & Claypool, 2012.

11 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Distributed computing by mobile robots: Solving the uniform circle formation problem. In *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS 2014)*, pages 217–232, 2014. doi:10.1007/978-3-319-14472-6_15.

12 Nao Fujinaga, Yukiko Yamauchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM Journal on Computing*, 44(3):740–785, 2015. doi:10.1137/140958682.

13 Marcello Mamino and Giovanni Viglietta. Square formation by asynchronous oblivious robots. In *Proceedings of the 28th Canadian Conference on Computational Geometry (CCCG 2016)*, pages 1–6, 2016.

14 Sheryl Manzoor, Samuel Sheckman, Jarrett Lonsford, Hoyeon Kim, Min Jun Kim, and Aaron T. Becker. Parallel self-assembly of polyominoes under uniform control inputs. *IEEE Robotics and Automation Letters*, 2(4):2040–2047, 2017. doi:10.1109/LRA.2017.2715402.

15 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.

16 Taichi Uehara, Yukiko Yamauchi, Shuji Kijima, and Masafumi Yamashita. Plane formation by semi-synchronous robots in the three dimensional euclidean space. In *Proceedings of the 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2016)*, pages 383–398, 2016. doi:10.1007/978-3-319-49259-9_30.

17 Masafumi Yamashita and Ichiro Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theoretical Computer Science*, 411:2433–2453, 2010. doi:10.1016/j.tcs.2010.01.037.

18 Yukiko Yamauchi, Taichi Uehara, Shuji Kijima, and Masafumi Yamashita. Plane formation by synchronous mobile robots in the three dimensional euclidean space. *Journal of the ACM*, 64(16), 2017. doi:10.1145/3060272.

19 Yukiko Yamauchi, Taichi Uehara, and Masafumi Yamashita. Brief announcement: Pattern formation problem for synchronous mobile robots in the three dimensional euclidean

space. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC 2016)*, pages 447–449, 2016. doi:10.1145/2933057.2933063.

20 Yukiko Yamauchi and Masafumi Yamashita. Pattern formation by mobile robots with limited visibility. In *Proceedings of the 20th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2013)*, pages 201–212, 2013. doi:10.1007/978-3-319-03578-9_17.

A Seventeen Symmetry Groups in 3D-space

We summarize the seventeen symmetry groups in 3D-space. Each rotation group is determined by rotation axes, mirror planes, and their arrangement. Our results also heavily rely on the order of each symmetry group and horizontal mirror planes. The following three tables show these properties together with typical polyhedra each of which is obtained as an orbit of a seed point in each symmetry group.

Table 2 Symmetry groups without rotation axis

	Symmetry	Order
C_1	Identity element	1
C_i	Point of inversion	2
C_s	Single mirror plane	2

Table 3 Symmetry groups with 2D rotation groups

	Principal axis	Other axes	Horizontal mirror	Order	Orbits
C_k	k-fold	-	N	k	
C_{kh}	k-fold	-	Y	$2k$	
C_{kv}	k-fold	-	N	$2k$	Pyramid with regular k-gon base
D_ℓ	ℓ-fold	ℓ 2-fold axes	N	2ℓ	
$D_{\ell h}$	ℓ-fold	ℓ 2-fold axes	Y	4ℓ	Hexagonal prism for D_6
$D_{\ell v}$	ℓ-fold	ℓ 2-fold axes	N	4ℓ	Hexagonal anti-prism for D_{6v}
S_m	m-fold	-	Y	m	Hexagonal anti-prism for S_{12}

Table 4 Symmetry groups with 3D rotation groups

	Rotation axes (folding)				Mirror planes	Order	Orbits
	2	3	4	5			
T	3	4	-	-	0	12	Snub tetrahedron
T_d	3	4	-	-	6	24	Regular tetrahedron
T_h	3	4	-	-	3	24	
O	6	4	3	-	0	24	Snub cube
O_h	6	4	3	-	9	48	Cube, regular octahedron
I	15	10	-	6	0	60	Snub icosahedron
I_h	15	10	-	6	15	120	Regular dodecahedron

Treasure Hunt with Barely Communicating Agents*

Stefan Dobrev[1], Rastislav Královič[2], and Dana Pardubská[3]

1 **Slovak Academy of Sciences, Bratislava, Slovakia**
2 **Comenius University, Bratislava, Slovakia**
3 **Comenius University, Bratislava, Slovakia**

Abstract

We consider the problem of fault-tolerant parallel exhaustive search, a.k.a. "Treasure Hunt", introduced by Fraigniaud, Korman and Rodeh in [13]: Imagine an infinite list of "boxes", one of which contains a "treasure". The ordering of the boxes reflects the importance of finding the treasure in a given box. There are k agents, whose goal is to locate the treasure in the least amount of time. The system is synchronous; at every step, an agent can "open" a box and see whether the treasure is there. The hunt finishes when the first agent locates the treasure.

The original paper [13] considers non-cooperating randomized agents, out of which at most f can fail, with the failure pattern determined by an adversary. In this paper, we consider deterministic agents and investigate two failure models: The failing-agents model from [13] and a "black hole" model: At most f boxes contain "black holes", placed by the adversary. When an agent opens a box containing a black hole, the agent disappears without an observable trace.

The crucial distinction, however, is that we consider "barely communicating" or "indirectly weakly communicating" agents: When an agent opens a box, it can tell whether the box has been previously opened. There are no other means of direct or indirect communication between the agents.

We show that adding even such weak means of communication has very strong impact on the solvability and complexity of the Treasure Hunt problem. In particular, in the failing agents model it allows the agents to be 1-competitive w.r.t. an optimal algorithm which does not know the location of the treasure, but is instantly notified of agent failures. In the black holes model (where there is no deterministic solution for non-communicating agents even in the presence of a single black hole) we show a lower bound of $2f+1$ and an upper bound of $4f+1$ for the number of agents needed to solve Treasure Hunt in presence of up to f black holes, as well as partial results about the hunt time in the presence of few black holes.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases parallel exhaustive search, treasure hunt, fault-tolerant search, weak coordination, black holes

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.14

1 Introduction

Searching is one of the most studied problems in computer science. As the search space is often huge, in order to speed up the search time a frequently used approach is to resort to parallel and/or distributed search. However, employing a large number of parallel searching agents gives rise to two important issues:

* This work has been supported by grant VEGA 2/0165/16.

- how to efficiently coordinate them, so that the agents complement each other's work instead of being an obstacle to each other
- how to deal with the inevitable failures, so that the search is successfully and efficiently completed and no area of search space is omitted.

On one hand, a tight coordination is desired, so that the agents can efficiently split their workload and react to behaviour/observations of other agents. On the other hand, the coordination itself incurs costs that might overwhelm any gains obtained by using more agents. This has motivated research into non-coordinated and weakly coordinated search agents, as lack of coordination brings inherent robustness and resilience to failures and tampering.

There are numerous variants of the search problem, differing in the structure of the search space, the power and the number of search agents, the communication mechanisms available to them and in the search space itself, the failure models, termination conditions as well as cost and payoff functions.

The particular variant of the search problem we are interested in has been introduced by Fraigniaud, Korman and Rodeh in [13]. It can be seen as an abstraction of a distributed exhaustive search as employed by BOINC [2] (for Berkeley Open Infrastructure for Network Computing), made famous for SETI@home. The goal is to exhaustively search the search space, until a desired item ("treasure", e.g. SETI signal, cryptographic key, ...) has been found. The search space is linear in nature, in the sense that there is a total ordering of the candidate solutions: given that a solution has not been found among the smaller candidates, the next candidate contains the most preferred solution, is the most likely to contain a solution, or is the least costly to check.

In this setting, a central server controls and distributes the work to volunteers. While typically there is a strong control over work tasks (which can be made of desired granularity and uniform size), there is very little control over the volunteer agents executing these work tasks. Therefore, it is highly desirable (for practical, security and anonymity reasons) to have no direct communication channels between the worker agents. Furthermore, for legacy and efficient implementation reasons, it is unfeasible to add environmental communication channels (e.g. whiteboards at the nodes of the search space). Hence, non-coordinated protocols are a very good match. It has been shown in [13] that by employing randomized agents, the cost of non coordination can be as low as a factor of 4.

Still, in this setting, there is a basic coordination mechanism which is always present, by the very nature of the setting: For each work task, the server knows whether it has been already solved or not. The central idea of this paper is to investigate how much can this essentially free information help in efficiently solving the treasure hunt problem.

2 Model, Outline and Related Work

Consider a Treasure Hunt (parallel linear search) problem: Given is a (potentially infinite) sequence of boxes, one of which contains a treasure. There are k deterministic agents in the system, each with its own unique ID from $\{1, \ldots, k\}$ [1]. Apart from the IDs, the agents are

[1] In fact, it is sufficient that the IDs are unique, and from a range $\{1, \ldots, \mathrm{MAXID}\}$, where MAXID is an a-priori known upper bound on a maximal ID. A preprocessing phase of cost $O(f \log k \log \mathrm{MAXID})$ (where f is an upper bound on the number of failures/black holes) can compact the IDs into the desired range, using standard parallel processing techniques. In order not to detract from the main focus of our paper, we prefer to assume this has already been done prior to commencing the treasure hunt.

identical, and have no means of communication. The system is synchronous and in each step each agent may open one box to check whether it contains the treasure. We consider only non-conflicting schedules[2], i.e. schedules with the property that, during any time step, each box is being accessed by at most one agent. When the box containing the treasure is opened, the hunt is finished. The goal is to minimize the hunt time.

In a reliable system, k agents can locate the treasure in $\lceil x/k \rceil$ steps, where x is the (unknown) location of the treasure: in i-th step, the agent j simply opens the box $k(i-1)+j$. Let us define the *speedup* as $\liminf_{x \mapsto \infty} x/T(k,x)$ where $T(k,x)$ is the hunt time with k agents, if the treasure is in box x. Hence, in a reliable system, k agents achieve a speedup of k. However, the just mentioned simple algorithm is very sensitive to errors: a failure of a single agent may lead to the treasure not being found at all. In this paper we study faulty systems; in particular we investigate treasure hunt under two failure models:

- in the *failing agents* model, also used in [13], at most $f < k$ agents may fail at any time during the computation. The *failure pattern* describing at what times which agents fail is chosen by an adversary.
- in the *black holes* model, we assume that at most f of the boxes contain a black hole. Whenever an agent opens a box with a black hole, the agent disappears without any trace. Again, the locations of the black holes are chosen by the adversary.

Consider first the failing agents model. Denote $T_f(k,x)$ the hunt time with k agents, taken as the worst case over all possible failure patterns in which up to f agents fail. Clearly, $f = k-1$ failing agents may be tolerated by a trivial algorithm in which every agent performs a sequential scan, having $T_f(k,x) = x + k$ [3]. However, this algorithm achieves only speedup of 1. Moreover, for $f = k-1$ essentially nothing better can be done, since the best achievable speedup for f failing agents is $k/(f+1)$ (see Claim 3.3).

The situation is even more bleak in the *black hole* model. Here even a single black hole makes the problem unsolvable in the deterministic setting (Claim 3.5). The proof of this heavily relies on the fact that the number of boxes is unbounded. If that is not true, the problem becomes solvable – Corollary 4 shows that treasure hunt can be solved by $O(\sqrt{n} \ln n)$ agents, where n is the number of boxes. The situation in randomized setting remains open.

The main contribution of this paper is to show that even a very limited means of communication have great impact on the solvability of the problem. The sole means of communication provided by the system is the following: When an agent opens a box, it learns whether this box has been opened before.

Even though the agents have no way to directly communicate, they can deduce the presence of faults from the presence of closed boxes which should have been opened by the agents that died. However, such boxes can be identified only by opening them, thus destroying any evidence; this read-once property makes algorithms in this setting rather tricky.

As we shall see shortly in Claim 3.6, even this indirect communication channel increases the power of the system: e.g. three agents can find the treasure in the presence of one black hole. The aim of this paper is to investigate the power of this indirect communication. In particular, the two central questions we try to answer are *"How fast can barely communicating agents find the treasure in the failing agents model?"* and *"How many black holes can be tolerated in the black hole model?"*.

[2] This simplifies arguments and technical issues; as we will see, the cost is negligible.
[3] The additive term is present because in order to ensure a non-conflicting schedule, the agents start their scans one after another.

For the first question, note that no speedup is possible in any of these models: indeed, in both failure models it is possible that $k-1$ agents die right at the beginning, leaving a single deterministic agent to do essentially the whole search. That's why we turn to competitive analysis, which is a well-established tool for analyzing similar types of adversarial settings, for the measurement of time efficiency: consider a setting where the agents are immediately notified about a failure (or, in the black holes model, when an agent enters a black hole). In this setting, there is a simple optimal algorithm which in each step assigns all k' living agents to the first k' unopened boxes. We compare the hunt time of a distributed algorithm to the hunt time of this optimal algorithm. In particular, the *competitive ratio* of an algorithm with k agents is $r_k = \limsup_{x\mapsto\infty} \sup_{\mathcal{F}} T_{\mathcal{F}}(k,x)/OPT_{\mathcal{F}}(k,x)$ where $\mathcal{F}$ is the *failure pattern* describing at what times which agents fail, x is the location of the treasure, and $T_{\mathcal{F}}(k,x)$ (resp. $OPT_{\mathcal{F}}(k,x)$) is the hunt time of the algorithm (resp. of the optimal algorithm). This measure is somewhat similar to the *non-coordination ratio* of Fraigniaud, Korman, and Rodeh [13]; the difference is that the latter considers the ratio of expected speedups, and as we have seen the speedup is not useful in the deterministic case. Also note that if we fix a failure pattern in which $k-1$ agents die at the beginning, the measure becomes directly related to speedup.

2.1 Our Results

We show that two agents (with the possibility that one of them fails) can achieve hunt time $T_{\mathcal{F}}(2,x) \leq OPT_{\mathcal{F}}(2,x) + O(\sqrt{x})$ (Claim 3.4). Since $OPT_{\mathcal{F}}(2,x) \geq x/2$, the algorithm is 1-competitive. This result is generalized to k agents with the possibility that $k-1$ of them fail in Theorem 2, in which case a hunt time $T_{\mathcal{F}}(k,x) \leq OPT_{\mathcal{F}}(k,x) + O(k^3\sqrt{x})$ can be achieved, again yielding a 1-competitive algorithm.

For the systems with black holes, we show that $2f+1$ agents are needed in order to guarantee a successful treasure hunt with f black holes (Theorem 6). We provide an upper bound in Theorem 8, showing that $4f+1$ agents are sufficient. The downside of the algorithm from Theorem 8 is that if the treasure is located at box x, boxes up to $O(xf^2)$ are used, making heavy use of the fact that the number of boxes is unlimited. At the expense of using $O\left((f \ln f)^2\right)$ agents, we develop in Theorem 9 an algorithm that uses only boxes up to $x + O\left((f \ln f)^2\right)$ in the case the treasure is in box x. This algorithm can be tweaked to work also for the setting where the number of boxes is finite (by having a set of agents dedicated to exploring the last $O\left((f \ln f)^2\right)$ boxes).

2.2 Related Work

The first work evaluating the search time as a function of the location of the searched item is [4, 5]. In this paper, which became a classic of online computing, the authors studied the *cow-path* problem with a single agent, and showed that locating the treasure (gate in their case) takes time $9d$, where d is the distance from the origin. As in our model, the search space is linear in nature and the search is for a single item. However, the principal difference is that unlike in our model in which the agents have direct access to boxes and pay unit price to access any box, in [5] the agent has to arrive to the box in order to search it, paying cost d to reach a box at distance d from its current location. In fact, in [5] authors consider several variants of the search problem, including searching in 2D and searching for large objects of known shape (lines, circles) but unknown location. [3] considers parallel search of large object by two robots, while the randomized version of the original cow-path problem was studied in [16]. The generalization of the cow path problem to many agents is studied in [7], where it is shown that independently of the number of agents, the group search cannot be performed faster than in time $9d - o(d)$.

A closely related research has been focused on ANTS problem, introduced in [12] and motivated by modeling ants foraging for food. The agents (ants) are randomized non-communicating Turing machines; the goal is to locate a treasure placed adversarially at some distance d from the origin. It is shown that n ants are able to achieve speedup of n, locating the treasure in time $O(d + d^2/n)$. Further work focused on limiting the power of the agents to finite machines [11] (but adding local communication), as well as considering agent failures [20]. In [21], the trade-off between the speedup and *selection complexity* of the ANTS problem is investigated, where the selection complexity is a measure of how likely the search strategy is to evolve in a natural environment. In [8], an infinite d-dimensional grid is explored by a group of probabilistic finite automata that can communicate only in a face-to-face manner. The aim is to find a hidden treasure, and the main question is how many agents are required to guarantee a finite expected hitting time. The destructive-read aspect of opening boxes in our model corresponds precisely to the ANTS model with a single pheromone. However, the crucial difference from our model is that ants have to pay the cost of travelling to the searched location, while in our model any box can be accessed directly at a unit cost.

Another closely related area of research deals with variants of the *do-all problem*: there are p processors that must collectively perform n tasks, such that each task is performed by at least one processor. The problem has been analyzed in a number of settings: synchronous processors with crashes (and possible restarts) under various modes of communication (full knowledge, message passing, etc.), asynchronous processors with adversarially limited rendezvous communication, etc. For a survey of results see [14].

Treasure hunt as a rendezvous between a searching agent and a static one has been studied in general graphs in [25, 26], in the context of universal traversal sequences. Finite search space, paying for edge traversal and a single searching agent make these results only tangentially relevant to our work. For a survey of results about the treasure hunt and rendezvous of two agents see also the book [1].

There is a long history of searching with failures in the classical Ulam-Renyi two players search game, where the failures represent incorrectness of the received answers. See [23] for a comprehensive survey of older results, and [15] for an example of more recent result in this area. Still, these works are only remotely related to our results.

In the black hole search problem introduced in [10, 9] (see [24] for a recent survey), some nodes of the network are black holes – nodes which destroy without observable trace any agent that enters them. The goal of the agents is to explore the network and locate all black holes. As in the ANTS problem, the agents pay for moving along the network, however, the primary focus is on establishing the number of agents necessary/sufficient to locate the black holes. Another difference from our setting is that the black holes search is considered solved only when all black holes have been identified, while in treasure hunt the hunt ends when the first agent locates the treasure.

The paper motivating our work and the one with the model closest to ours is [13], in which the authors show that k randomized non-cooperating agent achieve expected speedup of $k/4 + o(1)$ in solving the linear treasure hunt problem. In a recent follow-up [18], the authors investigate a variant of the problem, where the treasure is placed uniformly at random in one of a finite, large, number of boxes. They devise a non-coordinating algorithm that achieve a speed-up of $k/3 + o(1)$. In [19], an optimal algorithm for the case when the treasure is placed randomly with a known distribution is presented, and its running time is calculated for some specific distributions. The key differences are that we consider barely communicating deterministic agents and extend the results also to the black holes model of failures.

A somewhat related paper is [6], in which a similar problem is studied w.r.t. advice complexity: Given a number of n closed boxes, an adversary hides $k < \sqrt{n}$ items, each of unit value, within k consecutive boxes. The goal is to open exactly k boxes and gain as many items as possible.

3 Results

3.1 Building Blocks

As a point of interest, observe that given a long enough sequence of boxes guaranteed to not have been opened yet, the agents may implement any fault-tolerant message passing protocol; we present a general approach here. However, directly applying this approach on standard fault-tolerant message-passing protocols would generally result in too high search complexity. In the rest of the paper we use more efficient solutions specifically tailored to the problem at hand.

Let us consider the usual synchronous setting with k processors that start with some input values, and perform an r-round protocol, where in every round each processor receives messages from all other processors, performs some local computation, and sends messages to other processors. After r rounds, each processor has some output value. Moreover, a number of processors may crash during the protocol; in particular, a processor may also crash in the middle of the sending phase, in which case the messages to an arbitrary set of processors are delivered. Suppose that we have k agents that are assigned the same input values as the processors. We have the following observation:

▶ **Claim 3.1.** *Consider a message passing protocol with k processors and r rounds, that exchanges m-bit long messages, and tolerates $2f$ processor-crashes. This protocol can be implemented by k agents in $2k(r-1)(m+1)$ steps in the presence of $2f$ agent failures (or f black holes), provided that at the beginning the agents agree on a set of $(r-1)k^2(m+1)$ unopened boxes.*

Proof. The unopened boxes will be viewed as forming an array $M[t,i,j]$, where $t = 1,\ldots,r-1$ denotes the round, and $i,j = 1,\ldots,k$ are a pair of agents. The entry $M[t,i,j]$ consists of $m+1$ boxes that represent a message sent from processor i to processor j in t-th round. Each round t of the protocol is simulated as follows: at the beginning each agent j checks all entries $M[t-1,i,j]$, $i = 1,\ldots,k$ (if $t > 1$). If the last (i.e. $m+1$st) box from an entry is opened, it signals that the writing of $M[t-1,i,j]$ was successful, and in that case j considers the opened/unopened status of the first m boxes of $M[t-1,i,j]$ as bits of the message from i. Then agent j performs the local computation of processor j, and then writes values $M[t,j,i]$, $i = 1,\ldots,k$ based on the messages sent by the protocol. The last entry of $M[t,j,i]$ is opened to signal successful write. Both reading and writing of a round require $k(m+1)$ time steps, with the exceptions that there is no read phase in round 1, and no write phase in round r. So overall, the simulation takes $2k(r-1)(m+1)$ steps. Since no messages are sent in the last round, the overall number of boxes used by the communication is $(r-1)k^2(m+1)$.

Clearly, if an agent fails, it can be interpreted as a failure of a processor in the message-passing protocol. The last bit of $M[t,i,j]$ ensures that if an agent fails in the middle of writing an entry, incorrect message will not be considered.

On the other hand, note that every box is opened by at most two agents. Hence, a black hole can kill at most two agents, which leads to a crash of at most two processors. ◀

▶ Remark. Note that in the end, some boxes of $M[t,i,j]$ may remain unopened and hence need to be explicitly "cleaned" later on.

Observe that if in each round each processor communicates with at most one counterpart, the time complexity is $2r(m+1)$. Now we sketch how the simulation of message passing mechanism from the previous protocol can be used to compact the agents' IDs from a set $S = \{1 \dots, \text{MAXID}\}$ to $\{1, \dots k\}$.

- Let T be a complete binary tree with leaves corresponding to the set S. The leaves representing the ID of some agent are *assigned* to that agent, other leaves are *free*.
- Each internal vertex is assigned to the agent with the smallest ID in its subtree, or is free, if there is no such agent. Note that this assignment can be constructed level-by-level: for each internal vertex v there are at most two candidate agents. They don't know each other IDs, but may use a pre-determined place in the scratchpad (depending only on v) to exchange messages and find out the IDs. Hence, from that point on each agent knows which vertices are assigned to it.
- Use a message passing converge-cast in the tree (each agent simulates all vertices assigned to it) to collect the number of agents to the root. Along the way, collect also the number of active agents in each left and right subtree.
- Send this information back towards the leaves, where each agent computes its rank.
- Verify that each agent has computed its rank: Agent with rank i writes in row i of a $k \times k$ scratchpad and afterwards reads column i of the scratchpad. The number of opened boxes must equal the computed number of agents.
- If the verification fails (there must have been failure during the computation), repeat the whole process again.

As the depth of the tree is $\log \text{MAXID}$ and $f+1$ repetitions are sufficient to have a failure-free run, the computed values are at most k, and in each round each node communicates only with its children and parent, we obtain:

▶ **Claim 3.2.** *The processor labels can be compacted from* $\{1 \dots, MAXID\}$ *to* $\{1, \dots k\}$ *in* $O(f \log k \log MAXID)$ *time steps.*

Of course, compacting the IDs opens the boxes used in the scratchpads and can mess-up a subsequent treasure hunt algorithm. In some algorithms (e.g. for Theorem 8), it is sufficient to remember where the scratchpads used in the preprocessing end, so the new scratchpads needed use unopened boxes. Otherwise, a subroutine using non-communicating agents can be used to ensure opening of all boxes located in the pre-processing's scratchpads.

In the sequel we shall use the following *coordination problem* (actually a binary consensus on whether the agent 1 is alive) as a building block in more advanced algorithms: An unknown subset of the agents is alive. There is a distinguished leader agent all alive agents agree on (w.l.o.g.[4] agent 1); however, the leader might not be alive. All agents have to agree on a common boolean value, such that if 1 was dead at the beginning of the protocol, all agents agree on `false`, and if 1 was alive and there were no faults during the protocol, all agents agree on `true`. In view of Claim 3.1, we could use a well known simple consensus protocol (see e.g. [22]) to solve the problem in $O(k^2)$ steps using $O(k^3)$ boxes. However, there is a more efficient solution:

▶ **Lemma 1.** *The coordination problem can be solved in* $2k-2$ *steps, provided that the agents agree on a set of* $k(k-1)/2$ *unopened boxes.*

[4] If the agreed-upon leader is i, the agents cyclically assign themselves virtual IDs such that i gets 1: agent $(i+j) \bmod k$ gets virtual ID of $1+j \bmod k$.

Proof. The boxes will be viewed as an upper triangular matrix $M[i,j]$, $i = 1,\ldots,k; j = i+1\ldots,k$, such that open box $M[i,j]$ is a signal from agent i to agent j that agent 1 was alive at the beginning of the protocol.

Each agent has a boolean value that is initialized to `true` for agent 1, and `false` for other agents. The algorithm for each agent consists of two phases. In the *look* phase, agent $i > 1$ in step $i + r - 1$ opens box $M[r,i]$ for $r = 1,\ldots,i-1$. If it finds the box already opened, it sets its value to `true`. Afterwards, in the *shout* phase, an agent with value `true` opens box $M[i,r]$ in steps $i - 2 + r$ for $r = i+1,\ldots,k$. Note that the agent 1 (if alive) skips the look phase and proceeds directly to the shout phase.

If agent 1 is not alive at the beginning of the protocol, by a simple minimization argument, no agent can acquire value `true`: First, note that for every box $M[i,j]$, if the agent i has value `true`, it will open $M[i,j]$ before agent j opens it. Let t be the first time when some agent i obtains value `true`. To do so, it must have opened an already opened box $M[t-i,i]$, i.e. the agent $t-i$ already had a value `true` – a contradiction.

Further, if agent 1 is alive, and there are no faults, it successfully opens all boxes $M[1,i]$, and subsequently every agent j finds an open box at $M[1,j]$, acquiring value `true`.

Finally, we argue the agreement, i.e. if there is some surviving agent that finishes with value `true`, then all surviving agents finish with value `true`. Let i be the smallest[5] surviving agent that finished with value `true`. Then i opened boxes $M[i,j]$ for $j > i$ in its shout phase, and all larger surviving agents opened those boxes in their look phases and hence gained value `true`. ◀

3.2 Hunt Time in the Failing Agents Model

First let us observe that in systems without communication, the best possible speedup is $k/(f+1)$:

▶ **Claim 3.3.** *Consider a system with k non-communicating agents, in which at most $f < k$ agents may fail. There is an algorithm that achieves hunt time $T_f(k,x) \le x(f+1)/k + \binom{k-1}{f}$. Moreover, for any algorithm, $T_f(k,x) \ge x(f+1)/k$.*

Proof. For the upper bound, let $m = \binom{k}{f+1}$. We construct a schedule for m boxes and k agents, such that after $t = \binom{k-1}{f} = m(f+1)/k$ steps, every box is opened by $f+1$ distinct agents. The algorithm repeats this schedule in sequence for first, second, etc. m-tuple of boxes. Hence, if the treasure is somewhere in the i-th tuple (i.e. $\lceil x/m \rceil = i$), at least one agent opens the box with the treasure by the time

$$ti \le t\left(\frac{x}{m}+1\right) = x\frac{f+1}{k} + \binom{k-1}{f}.$$

What remains to be shown is how to construct the schedule. Consider a bipartite graph with partitions A, B, where A contains k vertices, and B contains $\binom{k}{f+1}$ vertices, corresponding to all $f+1$ element subsets of A. An edge connects every vertex from $v \in A$ with all vertices from B, such that v is in the corresponding set. Obviously, this graph has maximum degree $t = \binom{k-1}{f}$, and following König [17], it can be colored by t colors. This coloring is naturally interpreted as a schedule: an agent corresponding to a vertex from A opens a box corresponding to a vertex from B in time that is represented by the color of the edge.

[5] w.r.t. the virtual IDs

For the lower bound, the same argument works: let $x = \binom{k}{f+1} + 1$, and consider the first x boxes. Let $t = \binom{k-1}{f}$, and consider the actions of the k agents in a fault-free execution over the first t steps. Construct a t-colored bipartite graph of agents and boxes by connecting an agent i with box j by an edge with color c iff the agent opened the box in time step c. The graph has at most kt edges, so there must be a box $x' \leq x$ that is visited at most $\lfloor kt/x \rfloor \leq f$ times during the first t steps. Hence, if f agents may die, the adversary may delay the opening of x' to some time $t' > t$. Putting it together we have

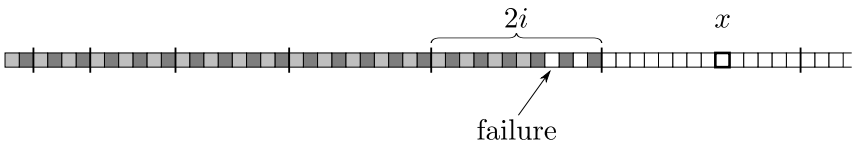
$$t' > t = \frac{(x-1)(f+1)}{k} \geq \frac{x'(f+1)}{k} - \frac{f+1}{k}$$

and the result follows, since $(f+1)/k$ is at most 1. ◀

However, even barely communicating agents can achieve competitive ratio of 1:

▶ **Claim 3.4.** *Two agents (with the possibility that one of them fails) can achieve hunt time* $T_{\mathcal{F}}(2,x) \leq OPT_{\mathcal{F}}(2,x) + O(\sqrt{\min\{t_1, x\}})$*, where* t_1 *is the time of the failure.*

Proof. The algorithm proceeds in phases, the aim of phase i is to open the next $2i$ boxes. The agent A_0 consecutively opens the even boxes, and the agent A_1 opens the odd boxes. At the end of the phase, each agent opens the last box that should have been opened by the other agent. If agent A_j finds an unopened box, it means that A_{1-j} died during the phase. A_j then goes back over the boxes that should have been opened by A_{1-j}, until it finds an opened box. A_j then proceeds sequentially from the beginning of the next phase.



Obviously, each box is eventually opened. What must be argued is how long the hunt may take compared to the optimal algorithm OPT. Note that in OPT, each agent that is alive opens a new box, and the opened boxes always form a consecutive interval. In order to obtain the result, we count the number of steps the algorithm behaves differently. First, before the failure occurs in phase i, there is one time step of delay per phase, in which both agents check each other. This accounts for at most $i = O(\sqrt{\min\{t_1, x\}})$ time steps. Finally, after the failure of one agent, the remaining agent may proceed to the end of the phase, and then return to find the treasure, accounting for another at most $2i = O(\sqrt{\min\{t_1, x\}})$ time steps. If the treasure is located beyond the phase in which the failure occurred, no additional delay is incurred. ◀

Now let us generalize the result to k agents. The main idea it the same as in the previous claim: proceed in phases of linearly increasing length, divide each phase among the surviving agents, then check at the end if some agents died during the phase, and if so, finish their part. However, care must be taken, because now also an agent finishing the part of an already failed agent, may fail. Also, in order to divide the work among themselves, the agents need to know the number of surviving agents, which is not always possible. We can formulate the following theorem:

▶ **Theorem 2.** *k agents (with the possibility that $k-1$ of them fails) can achieve hunt time* $T_{\mathcal{F}}(k,x) \leq OPT_{\mathcal{F}}(k,x) + O(k^3\sqrt{x})$.

Algorithm 1 Algorithm for one phase.

```
1: procedure PHASE(i)
2:     if i > 1 ∧ S_i ≠ S_{i−1} then
3:         all agents check all k_{i−1}d_{i−1} boxes from phase i − 1
4:     end if
5:     divide the k_i d_i boxes among the k_i agents
6:     each agent checks the assigned d_i boxes
7:     compute set S_{i+1} using k^3 boxes from phase i + 1
8: end procedure
```

Proof. The algorithm proceeds in phases. Each phase i starts with a set $S_i \subseteq \{1, \ldots, k\}$, such that all agents agree on S_i, and S_i contains all agents that are alive at the beginning of phase i. Initially $S_1 = \{1, \ldots, k\}$. Let $k_i = |S_i|$, and $d_i = k^3 i$. The aim of phase i is to check $k_i d_i$ boxes.

The $k_i d_i$ boxes are divided among the k_i agents such that each agent is responsible for one class of boxes modulo k_i. Agents spend d_i time steps to check the assigned class. However, some agent may have died during the phase, and some of them might have already been dead at the beginning. At the end of the phase, the agents agree on the new set S_{i+1} as follows. Each agent $j \in S_i$ in turn uses the consensus algorithm from Lemma 1, using a fresh set of k^2 boxes from the next phase. If the result of the consensus algorithm is `true`, j is included in the set S_{i+1}. If, at the beginning of phase $i + 1$, $S_i = S_{i+1}$, it means that all agents from S_i were alive when the consensus computation started, and so all boxes from phase i have been checked. If, on the other hand $S_{i+1} \neq S_i$, some agents from S_i died. It might have been during the consensus computation, but in any case, the boxes of phase i must be rechecked. In order to avoid the necessity to argue about further failures, all agents from S_{i+1} recheck all boxes from phase i. Since at least one agent is guaranteed to survive, all boxes from phase i will be opened.

To argue about the speedup, we again count the number of time steps in which the algorithm behaves differently from OPT. First, there are $O(k^2)$ time steps to compute the set S_{i+1} in each phase. Since there are $O(\sqrt{x})$ phases, this account for $O(k^2\sqrt{x})$ steps. Also, there are at most $O(k)$ failures, and each failure can trigger at most one recheck of a phase.

Since a phase contains at most $O(k^2\sqrt{x})$ boxes, this accounts for another $O(k^3\sqrt{x})$ steps. ◀

3.3 Number of Agents in Black Holes Model

While in the failing agents model even a failure of $k - 1$ agents can be tolerated without communication, in the black holes mode a single black hole can prevent any deterministic non-communicating algorithm from finding the treasure:

▶ **Claim 3.5.** *Consider a system with k non-communicating agents with a single black hole. There is no algorithm that always successfully finds the treasure.*

Proof. For a box given x, let the *signature* of x, $\tau_x = [t_{x,1}, \ldots, t_{x,k}]$ be the k-tuple such that $t_{x,j}$ is the first time that agent j opens x in a fault-free execution; if j never opens x, then $t_{x,j} = \infty$. If there are two boxes x, x' such that $\tau_x \leq \tau_{x'}$, i.e. for all j $t_{x,j} \leq t_{x',j}$, then placing a black hole in x and treasure in x' results in the algorithm never finding the treasure. To conclude, it is sufficient to find two such boxes. Let us call a *pattern* of a signature τ_x the set of positions j for which $\tau_{x,j} = \infty$. Since there is a finite number of patterns (namely,

less than 2^k), and the number of boxes in unlimited, there is a pattern p that is shared by infinitely many boxes. Let us fix a box b with pattern p, and count the maximum number of boxes x (with pattern p) for which $\tau_x \not\geq \tau_b$. For any such box x there must be some index j such that $t_{x,j} < t_{b,j}$, and $t_{b,j}$ is finite. However, for any (finite) integer d there is at most one box x such that $t_{x,j} = d$. Hence, there are at most finitely many boxes (with pattern p) such that $\tau_x \not\geq \tau_b$. Since the number of boxes with pattern p is unlimited, there is a box x for which $\tau_x \geq \tau_b$. ◀

At the core of the proof of the previous claim is the fact that the sequence of boxes is potentially unlimited. If there is an upper bound on the number of boxes, the situation is much different.

▶ **Lemma 3.** *For large enough z, let $x = z^4$. Boxes $[x] = \{1, 2, \ldots, x\}$ can be cleared in z^3 steps by $g(x)$ agents without communication in the presence of at most z black holes, where $g(x) = O(z^2 \ln z)$.*

▶ **Corollary 4.** *For $n \geq f^4$, $O(\sqrt{n} \ln n)$ non-communicating agents can perform treasure hunt in the presence of f black holes.*

▶ **Corollary 5.** *In the algorithm from Lemma 3, every box e is scheduled to be opened by at least $z + 1$ agents.*

Let us focus now on the barely communicating model. We start by observing that enhancing the model with communication indeed makes it stronger, as it can now cope with a single black hole:

▶ **Claim 3.6.** *Three agents can find the treasure in the presence of one black hole.*

Proof. The first agent, A, starts by sequentially opening the boxes, i.e. A opens box i at time i, $i = 1, 2, \ldots$. It is followed by two agents, B_1, B_2 opening even and odd boxes, respectively, i.e. B_j opens box $2i + 1 - j$ at time $2i + 1$, $i = 1, 2, \ldots$. If no box contains a black hole, B_1, B_2 are always behind A, and see their boxes had already been opened. So when there is a black hole, A finds it first, and dies there, followed by one of the agents B_1, B_2. The other B agent survives, and finds an unopened box, z, in the next step. Then it knows that the black hole is located in $z - 1$, so it can finish the search sequentially. ◀

Let us note that each black hole can kill at least two agents, giving us a lower bound on the number of agents:

▶ **Theorem 6.** *$2f + 1$ agents are needed to perform treasure hunt in the presence of f black holes.*

Proof. For the sake of contradiction consider a treasure hunt algorithm with at most $2f$ agents. We construct a configuration of f black holes such that in finite time all $2f$ agents will be killed. Since the number of boxes is unlimited, if the treasure is located far enough, it will not be found.

The configuration of black holes will be constructed in phases. First, consider the computation C_0 of the algorithm in a configuration without black holes. There must be a box that is checked by at least two agents: if every box is checked by at most one agent, then there must be an agent that is the only one to check at least two boxes. By placing a black hole in the first one, and the treasure in the second one, we have a configuration in which the algorithm fails to find the treasure. Consider the first time t_1 when an agent checks an already opened box x_1. Construct a computation C_1 by placing a black hole in box x_1. We

claim that up to time t_1, the computations C_0, and C_1 are identical: indeed the agents are deterministic, and up to time t_1 each of them opens a yet unopened box. In C_1, the black hole at x_1 killed two agents, the second one at time t_1. Let S_1 be the set of unopened boxes at t_1, and repeat the same argument. There must be a box from S_1 that is opened by two agents in C_1 (after time t_1). Construct a computation C_2 by placing a black hole in the first box x_2 from S_1 that is visited by a second agent at time t_2. Up to t_2, C_1 and C_2 are identical: let t_2', $t_1 < t_2' < t_2$, be the time x_2 is first opened by an agent A. Clearly, C_1 and C_2 are identical up to t_2'. After t_2', agent A in C_1 either opened boxes outside S_1 that were already opened anyway, or another boxes from S_1 that were never opened by another agent (up to t_2). In any case, the disappearance of A in C_2 has no effect on the set of opened boxes in C_2. Hence, the black hole kills another two agents. Continuing this way, we find a set of f black holes that kill all $2f$ agents. ◀

We now proceed to showing an asymptotically matching upper bound of $4f + 1$. The idea is to sequentially scan the boxes in phases, where i-th phase ensures that the i-th box is opened. In order to do so, the agents elect a leader that probes the box, and let the others know if it survived, using a fresh set of boxes. However, if the agents agree that the leader is not alive, they still can not be sure whether the i-th box contains a black hole, since the leader election algorithm itself might have failed. Hence, the i-th box must be checked once more, maybe requiring several iterations in a single phase. However, care must be taken that if the i-th box is indeed a black hole, not too many agents die there. First, let us formulate the *leader election* algorithm that will be employed:

▶ **Lemma 7.** *Using a set of k fresh boxes, the agents may perform an algorithm, such that*
1. *At most one agent declares itself a leader.*
2. *If the k boxes do not contain a black hole, exactly one leader is elected.*

Proof. In the first step, agent i opens the i-th box. Then it scans the subsequent boxes, until it either finds an opened box, or reaches the last box. In the latter case, the agent is declared a leader. ◀

▶ **Theorem 8.** *$4f + 1$ agents are sufficient to find the treasure in the presence of f black holes.*

Proof. The algorithm works in phases, such that at the end of phase i, boxes $\{1, \ldots, i\}$ (and possibly some others) are opened. A phase to check box x consists of several iterations. Each iteration considers a fresh set of $k^2 + k + 1$ boxes, called a *scratchpad*, that are used for communication, and have the following structure: there are two sets of k boxes to perform a leader election algorithm from Lemma 7, two sets of $k(k-1)/2$ boxes to perform a coordination algorithm from Lemma 1, and one box called *survival flag*. The iteration starts by the agents performing a leader election. Subsequently, all agents except the leader perform another leader election algorithm to elect a *deputy*. Next, the leader opens the survival flag box, and immediately proceeds to check x, after which it performs the coordination algorithm from Lemma 1 to signal the other agents. If the result of the coordination is `true`, both the iteration, and the phase are ended, and a new phase starts to check the next box. If the result of the coordination is `false`, the deputy checks the survival flag. If it was opened, the deputy uses the second coordination algorithm to inform the other agents. If the result of the second coordination is `true`, again, both the iteration, and the phase are finished. If the result is `false`, new iteration of the same phase starts.

First, we show that at the end of a phase, box x was opened. The phase ends only when at least one of the coordination algorithms results in `true`. The first one may result in `true`

only when the leader survives the visit of box x. The second one only if the deputy finds the survival flag open and survives to tell that to others. Hence, the leader also survives opening the survival flag and as it immediately proceeds to open box x, x was definitively opened.

Second, let us note that if the scratchpad does not contain a black hole, the corresponding iteration is the last iteration of a phase. Indeed, if there is no black hole in the scratchpad, both leader, and deputy are successfully elected. The leader then successfully opens the survival flag, and proceeds to open x. If it survives, it again successfully performs the coordination, and phase ends. If the leader dies in x, the deputy reads the survival flag, and successfully signals the others.

Finally, let us count how many agents may die. In any given iteration, agents may die either in the scratchpad or in the box being checked, x. The scratchpads use fresh boxes in each iteration, and each box in the scratchpad is accessed by at most two different agents. Hence, a single black hole may kill at most two agents in a scratchpad. Note that the particular box with the black hole will be checked at some later phase (playing the role of x) where it may kill additional agents; these will be accounted for separately. Overall during the whole computation, at most $2f$ agents die in scratchpads.

Now let us count the agents that die while entering the checked box x. Obviously, there must be a black hole located in x, so there are at most f such boxes, each of them killing several agents over a number of iterations. However, as argued above, an iteration without a black hole in the scratchpad ends the phase. As there are, overall, at most f iterations with a black hole in the scratchpad, at most $2f$ agents die when entering the checked box.

Altogether, at most $4f$ agents may die. ◀

While the previous theorem showed that $4f + 1$ agents are sufficient, it relied heavily on the fact that the number of boxes is unlimited. Indeed, in order to locate the treasure in box x, boxes up to $O(xf^2)$ were used. A natural scenario, however, is that the number of boxes is finite. The algorithm from the previous theorem can not be used in this case, since it would not be able to find a treasure located in the last part of the boxes. In order to remedy this situation, we propose another algorithm that uses more agents (in particular, $O\left((f \ln f)^2\right)$ of them), but uses only boxes up to $x + O\left((f \ln f)^2\right)$ to locate the treasure in box x. Although omitted in this paper, it is possible to adjust the algorithm to the setting with finite number of boxes (by using a few additional agents).

The first idea is to use the algorithm from Lemma 3 to clean the scratchpad after each iteration. However, it can not be used in a straightforward manner, since the size of the scratchpad is quadratic in the number of agents, and Lemma 3 needs at least $\Omega(\sqrt{n} \ln n)$ agents to clean it. Therefore we shall implement a similar idea more carefully.

▶ **Theorem 9.** *$O(f^2 \ln^2 f)$ agents can locate the treasure in the presence of f black holes, even if the number of boxes is limited, provided that it is at least $x + O(f^2 \ln^2 f)$ where x is the location of the treasure.*

The algorithms from Theorems 8 and 9 are very slow, in fact much slower than a single searching agent in absence of black holes. Hence, the question is: What kind of speedup can be achieved in the presence of black holes, given sufficiently many agents?

As we see in the following theorem, a single black hole does not pose much of a problem:

▶ **Theorem 10.** *In the presence of one black hole, $k > 2$, k even, agents can locate the treasure in time $T(k, x) \leq x/(k-2) + O(k + \sqrt{x}) \leq OPT(k-2, x) + O(k + \sqrt{x})$.*

Proof. Divide the sequence of boxes into k' classes modulo $k' = (k-2)/2$ called *rows*. Each row is divided into *blocks*, with i-th block being of length $2(k' + i)$. Two *explorer* agents LE_r and RE_r are assigned to each row. In addition, there are two *checking* agents: LC and RC.

Let $s_{r,i}$, $t_{r,i}$ be the first and the last box of i-th block of row r. The algorithm works in phases; the aim of each phase is to clear one block in each row. At the beginning of phase i, the explorer agents in each row scan the block in opposite directions, i.e. the agent LE_r opens boxes starting by $s_{r,i}$ and finishing just before $t_{r,i}$ (i.e. without touching $t_{r,i}$), while RE_r symmetrically starts at $t_{r,i}$ and proceeds leftwards. If an explorer opens an already opened box, its task in the current phase is finished; it then proceeds to the next phase (to clear the next block). Let $T_i = 1 + \sum_{j=1}^{i-1}(k' + j + 1)$; note that without any black hole, all explorers start phase i in time step T_i.

The activity of the checking agents in block i is also counted in the phase i: The left checking agent LC scans the $s_{r,i}$ boxes. If all of them have already been opened, it becomes RC for the next phase. However, if it finds an unopened box in row r, it continues as LE_r in the current phase, and onwards. The RC agent works in a symmetrical way. Moreover, the checking agents make sure not to start any activity in block i before time $T_i + 1$; this ensures that they do not to interfere with the explorers.

First, let us consider an execution without a black hole. In this case, the explorers are always ahead of checkers (who don't play any role in the analysis yet), and they start exploring each block i at time T_i. In each phase, all explorers open a new box in every step. The only exception is the last step of the phase, when already opened boxes are probed by all explorers. Hence, if the treasure is located in a box x in i-th block, the delay w.r.t. $OPT(k-2, x)$ is at most the number of blocks (i.e. $i-1$) plus the length of the block (i.e. $2(k'+i)$). As $x \in \Theta((k+i)^2)$, the theorem holds.

Now let us consider a black hole located in a row r. All other rows are unaffected, and proceed as in the case without black holes. Moreover, if the treasure is found before the black hole, again, the analysis of the case without black holes applies. Let us focus only on row r, and let us assume that the black hole is encountered before the treasure. If the black hole is in the interior of the block (i.e. not the first or last box), both explorer agents die, with no unopened boxes left in this block; both checkers find unopened boxes in the next phase, become corresponding explorers for row r, and the algorithm proceeds without further delays and faults. If the black hole is in the first box of row r at phase i, both LE_r and LC die there. RE_r checks the whole block i of row r, and proceeds to the next phase, as well as RC. However, RC continues as LC in phase $i+1$, detects unopened first box and becomes LE_r. Analogously, if the black hole is in the last box of a block, the right explorer and checker die there, and the left checker eventually becomes the right explorer for the rest of the computation.

Note that the presence of the black hole in row r delays the exploration of row r by the delay between the time the explorer would have started exploring the block, and the time the checker checks whether it has indeed done so. As mentioned above, this delay is at most $r + i$. ◀

Observe that the technique used in the proof of Theorem 10 cannot be extended to more black holes. In fact, it is unclear what kind of speedup is possible in the presence of up to f black holes.

4 Conclusions and Open Problems

We have shown that adding even very weak indirect coordination (namely, the ability to detect whether a box has been previously opened, opening it and destroying the information in the process) allows deterministic agents to solve treasure hunt (a) with competitive ratio 1 in the failing agents models, and (b) with asymptotically optimal number of agents in the

black hole model. This is somewhat surprising, as this is in some sense the weakest natural form of coordination.

Several research questions remain open:

- what is the hunt time in the black hole model? We have some partial results for 1 and 2 black holes, but not a general solution. It is not clear whether it is possible to have speedup of $\Omega(k - f)$.
- how to solve treasure hunt with black holes using randomized non-coordinated agents? How many agents are needed and what would be the speedup? What about allowing both weak coordination and randomization?
- our solutions are not robust w.r.t. to timing and ordering of boxes. How do you make them robust and at what cost?
- what is the weakest coordination (a) that still allows competitive ratio of 1 in the failing agents model? (b) that allows to solve treasure hunt by deterministic agents?

Acknowledgement. We thank the anonymous referee for pointing out several inaccuracies in the manuscript.

References

1. Steve Alpern and Shmuel Gal. *The Theory of Search Games and Rendezvous*. International Series in Operations Research & Management Science. Springer US, 2006.
2. David P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, 2004. URL: `http://boinc.berkeley.edu`.
3. Ricardo Baeza-Yates and René Schott. Parallel searching in the plane. *Computational Geometry*, 5(3):143–154, 1995. `doi:10.1016/0925-7721(95)00003-R`.
4. Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. *Searching with uncertainty extended abstract*, pages 176–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988. `doi:10.1007/3-540-19487-8_20`.
5. Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J.E. Rawlins. Searching in the plane. *Inf. Comput.*, 106(2):234–252, 1993. `doi:10.1006/inco.1993.1054`.
6. Hans-Joachim Böckenhauer, Juraj Hromkovic, Dennis Komm, Richard Královic, and Peter Rossmanith. On the power of randomness versus advice in online computation. In Henning Bordihn, Martin Kutrib, and Bianca Truthe, editors, *Languages Alive - Essays Dedicated to Jürgen Dassow on the Occasion of His 65th Birthday*, volume 7300 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2012. `doi:10.1007/978-3-642-31644-9_2`.
7. Marek Chrobak, Leszek Gąsieniec, Thomas Gorry, and Russell Martin. *Group Search on the Line*, pages 164–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. `doi:10.1007/978-3-662-46078-8_14`.
8. Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an infinite space with finite memory scouts. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 207–224. SIAM, 2017. `doi:10.1137/1.9781611974782.14`.
9. Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Searching for a black hole in arbitrary networks: optimal mobile agents protocols. *Distributed Computing*, 19(1):1–99999, 2006. `doi:10.1007/s00446-006-0154-y`.
10. Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Mobile search for a black hole in an anonymous ring. *Algorithmica*, 48(1):67–90, 2007. `doi:10.1007/s00453-006-1232-z`.

11 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. *Solving the ANTS Problem with Asynchronous Finite State Machines*, pages 471–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-43951-7_40.

12 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative search on the plane without communication. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 77–86, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332444.

13 Pierre Fraigniaud, Amos Korman, and Yoav Rodeh. Parallel exhaustive search without coordination. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 312–323. ACM, 2016. doi:10.1145/2897518.2897541.

14 Chryssis Georgiou. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer Publishing Company, Incorporated, 1st edition, 2010.

15 Nicolas Hanusse, Dimitris J. Kavvadias, Evangelos Kranakis, and Danny Krizanc. Memoryless search algorithms in a network with faulty advice. *Theor. Comput. Sci.*, 402(2-3):190–198, 2008. doi:10.1016/j.tcs.2008.04.034.

16 Ming-Yang Kao, John H. Reif, and Stephen R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 441–447, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=313559.313848.

17 Dénes König. Über graphen und ihre anwendung auf determinantentheorie und mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916. doi:10.1007/BF01456961.

18 Amos Korman and Yoav Rodeh. Parallel linear search with no coordination for a randomly placed treasure. *CoRR*, abs/1602.04952, 2016. arXiv:1602.04952.

19 Amos Korman and Yoav Rodeh. Parallel search with no coordination. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017*, page to appear, 2017.

20 Tobias Langner, Jara Uitto, David Stolz, and Roger Wattenhofer. *Fault-Tolerant ANTS*, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-662-45174-8_3.

21 Christoph Lenzen, Nancy Lynch, Calvin Newport, and Tsvetomira Radeva. Searching without communicating: tradeoffs between performance and selection complexity. *Distributed Computing*, pages 1–23, 2016. doi:10.1007/s00446-016-0283-x.

22 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

23 Andrzej Pelc. Searching games with errors - fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, 2002. doi:10.1016/S0304-3975(01)00303-6.

24 Mengfei Peng, Wei Shi, Jean-Pierre Corriveau, Richard Pazzi, and Yang Wang. Black hole search in computer networks: State-of-the-art, challenges and future directions. *Journal of Parallel and Distributed Computing*, 88:1–15, 2016. doi:10.1016/j.jpdc.2015.10.006.

25 Amnon Ta-Shma and Uri Zwick. Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms*, 10(3):12:1–12:15, 2014. doi:10.1145/2601068.

26 Qin Xin. Faster treasure hunt and better strongly universal exploration sequences. *CoRR*, abs/1204.5442, 2012. arXiv:1204.5442.

Anonymous Processors with Synchronous Shared Memory: Monte Carlo Algorithms*

Bogdan S. Chlebus[1], Gianluca De Marco[2], and Muhammed Talo[3]

1 **Department of Computer Science and Engineering, University of Colorado Denver, Denver, Colorado 80217, USA**
bogdan.chlebus@ucdenver.edu

2 **Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, 84084 Salerno, Italy**
demarco@dia.unisa.it

3 **Bilgisayar Mühendisliği, Munzur Üniversitesi, 62000 Tunceli, Turkey**
muhammedtalo@munzur.edu.tr

Abstract

We consider synchronous distributed systems in which processors communicate by shared read-write variables. Processors are anonymous and do not know their number n. The goal is to assign individual names by all the processors to themselves. We develop algorithms that accomplish this for each of the four cases determined by the following independent properties of the model: concurrently attempting to write distinct values into the same shared memory register either is allowed or not, and the number of shared variables either is a constant or it is unbounded. For each such a case, we give a Monte Carlo algorithm that runs in the optimum expected time and uses the expected number of $\mathcal{O}(n \log n)$ random bits. All our algorithms produce correct output upon termination with probabilities that are $1 - n^{-\Omega(1)}$, which is best possible when terminating almost surely and using $\mathcal{O}(n \log n)$ random bits.

1998 ACM Subject Classification C.1.4 Parallel Architectures, F.1.1 Models of Computation, F.1.2 Modes of Computation

Keywords and phrases anonymous processors, synchrony, shared memory, read-write registers, naming, Monte Carlo algorithms

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.15

1 Introduction

We study naming algorithms in distributed systems consisting of anonymous processors that communicate by reading from and writing to shared memory. We say that a parameter of an algorithmic problem is *known* when it can be used in a code of algorithm. We restrict our attention to synchronous systems and do not assume that the number of processors n is known.

The model of synchronous systems with read-write registers is known as the Parallel Random Access Machine (PRAM). It is a generalization of the Random Access Machine model of sequential computation [16] to the realm of synchronous concurrent processing.

We consider two categories of naming problems depending on how much shared memory is available for a PRAM. In one case, a constant number of memory cells is available. This means that the amount of memory is independent from the number of processors n but as

* The full version of this paper merged with [12] is available as [13], https://arxiv.org/abs/1507.02272.

Table 1 Four naming problems, as determined by the PRAM model and the available amount of shared memory, with the respective performance bounds of their solutions as functions of the number of processors n. When time is marked as "polylog" this means that the algorithm comes in two variants, such that in one the expected time is $\mathcal{O}(\log n)$ and the amount of used shared memory is suboptimal $n^{\mathcal{O}(1)}$, and in the other the expected time is suboptimal $\mathcal{O}(\log^2 n)$ but the amount of used shared memory misses optimality by at most a logarithmic factor.

PRAM Model	Memory	Time	Algorithm
Arbitrary	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Arbitrary-Bounded-MC in Section 3
Arbitrary	unbounded	polylog	Arbitrary-Unbounded-MC in Section 4
Common	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	Common-Bounded-MC in Section 5
Common	unbounded	polylog	Common-Unbounded-MC in Section 6

large as needed in an algorithm's design. In the other case, the amount of shared memory cells is unlimited, and how much is used by an algorithm depends on n. When an unbounded amount of memory cells is assumed to be available, then the expected number of memory cells that are actually used is considered as a performance metric.

Independently of the amount of shared memory available, we consider the two versions of the naming problems that are determined by the semantics of concurrent writing. This is represented by the corresponding PRAM variants, which are either the Arbitrary PRAM or the Common PRAM.

Naming in the PRAM model is also considered in a companion paper [12], which is about the same problem to assign names for the anonymous processors, with the only difference that n is assumed to be known in [12], while we assume in this paper that n is unknown. Whether n is known or not is reflected in the classes of algorithms we develop in these two papers: these are Monte Carlo algorithms for an unknown n in this paper, and Las Vegas algorithms for a known n in [12].

A summary of the results

We consider four naming problems in synchronous read-write shared memory. They are determined by two independent specifications the naming problems: the amount of shared memory and the PRAM's variant.

The naming algorithms we give terminate with probability 1 and are all Monte Carlo. Each algorithm uses the optimum expected number $\mathcal{O}(n \log n)$ of random bits. We show that a Monte Carlo naming algorithm that uses $\mathcal{O}(n \log n)$ random bits has to have the property that it fails to assign unique names with the probability that is $n^{-\Omega(1)}$. All Monte Carlo algorithms that we give have the optimum polynomial probability of error. The list of the naming problems' specifications and the corresponding algorithms with their performance bounds are summarized in Table 1. Most proofs are omitted; they can be found in [13].

Previous and related work

The naming problem for a synchronous PRAM has not been previously considered in the literature, to the best of the authors' knowledge, except for the companion paper [12]. The problem of concurrent communication in anonymous networks was first considered by

Angluin [3]. That work showed, in particular, that randomization is needed in naming algorithms when executed in environments that are perfectly symmetric; other related impossibility results are surveyed by Fich and Ruppert [18]. There is a voluminous literature on various aspects of computing and communication in anonymous systems, we concentrate on the related topics for anonymous *asynchronous* distributed shared-memory systems.

We begin with work on naming in shared-memory systems with read-write registers. Lipton and Park [23] considered naming in asynchronous distributed systems with read-write shared memory controlled by adaptive schedulers; they proposed a solution that terminates with positive probability, and which can be made arbitrarily close to 1 assuming that n is known. Eğecioğlu and Singh [15] proposed a polynomial-time Las Vegas naming algorithm for asynchronous systems with known n and read-write shared memory with oblivious scheduling of events. Kutten et al. [22] provided a thorough study of naming in asynchronous systems of shared read-write memory. They gave a Las Vegas algorithm for an oblivious scheduler for the case of known n, which works in the expected time $\mathcal{O}(\log n)$ while using $\mathcal{O}(n)$ shared registers, and also showed that a logarithmic time is required to assign names to anonymous processes. Additionally, they showed that if n is unknown then a Las Vegas naming algorithm does not exist, and a finite-state Las Vegas naming algorithm can work only for an oblivious scheduler. Panconesi et al. [24] gave a randomized wait-free naming algorithm in anonymous systems with processes prone to crashes that communicate by single-writer registers. The model considered in that work assigns unique single-writer registers to nameless processes and so has a potential to defy the impossibility of wait-free naming for general multi-writer registers proved by Kutten et al. [22]. Buhrman et al. [11] considered the relative complexity of naming and consensus problems in asynchronous systems with shared memory that are prone to crash failures, demonstrating that naming is harder that consensus.

Now we review work on problems in anonymous distributed systems different from naming. Aspnes et al. [4] gave a comparative study of anonymous distributed systems with different communication mechanisms, including broadcast and shared-memory objects of various functionalities, like read-write registers and counters. Alistarh et al. [2] gave randomized renaming algorithms that act like naming ones, in that process identifiers are not referred to; for more or renaming see [1, 6, 14]. Aspnes et al. [5] considered solving consensus in anonymous systems with infinitely many processes. Attiya et al. [7] and Jayanti and Toueg [21] studied the impact of initialization of shared registers on solvability of tasks like consensus and wakeup in fault-free anonymous systems. Bonnet et al. [10] considered solvability of consensus in anonymous systems with processes prone to crashes but augmented with failure detectors. Guerraoui and Ruppert [19] showed that certain tasks like time-stamping, snapshots and consensus have deterministic solutions in anonymous systems with shared read-write registers prone to process crashes. Ruppert [25] studied the impact of anonymity of processes on wait-free computing and mutual implementability of types of shared objects.

A systematic exposition of shared-memory algorithm can be found in [8], when approached from the distributed-computing perspective, and in [20], when approached from the parallel-computing one. General questions of computability in anonymous message-passing systems implemented in networks were studied by Boldi and Vigna [9], Emek et al. [17], and Sakamoto [26].

2 Technical Preliminaries

Two operations are said to be performed concurrently when they are invoked in the same round of an execution of a PRAM. We assume that concurrent reading from a memory cell and writing to this same memory cell never occur. This can be made without loss of generality for a synchronous PRAM because we can partition an execution into alternating "writing" and "reading" rounds, which results in slowing the execution by at most a factor of 2. The meaning of concurrent reading from the same memory cell is straightforward, in that all the readers get the value stored in this memory cell.

Concurrent writing to the same memory location needs to be further clarified.

When multiple writers want to write the same value each in the same round to the same memory cell, then we should assume that this value gets written. This scenario is so enticing, that it leads to a PRAM variant called *Common*, for which it is assumed that only such concurrent writes are legitimate, in that an attempt to write different values concurrently to the same memory location results in a runtime error.

On the other hand, not having to worry about consistency of written values is also attractive, which leads to a PRAM variant called *Arbitrary*. This model allows any admissible value to be attempted to be written concurrently. A downside is that the model does not determine the outcome of a write but only that one of the values gets written. A consequence is that when we argue about correctness then all possible selections among the attempted values as actually written successfully need to be considered.

Balls into bins

In the course of probabilistic analysis of algorithms, we will often model actions of processors by throwing balls into bins. This can be done in two natural ways. One is such that memory addresses are interpreted as bins and the values written represent balls, possibly with labels. Then total number of balls considered will always be n, that is, be equal to the number of processors of a PRAM. Another possibility is when bins represent rounds and selecting a bin results in performing a write to a suitable shared register in the respective round.

Throwing balls into bins will be performed repeatedly in each instance of modeling the behavior of an algorithm. Each instance of throwing a number of balls into bins is then called a *stage*. There will be an additional numeric parameter $\beta > 0$, and we call the process of throwing balls into bins the *β-process*, accordingly. This parameter β may determine the number of bins in a stage and also when a stage is the last one in an execution of the β-process.

When we sum up the numbers of available bins over all the stages of an execution of a β-process until termination, then the result is *the number of bins ever needed* in this execution. Similarly, *the number of bits ever generated* in an execution of a β-process is the sum of all the numbers of random bits needed to be generated to place balls, over all the stages and balls until termination of this execution.

Verifying collisions

We will use a randomized procedure for Common PRAM to verify if a collision occurs in a bin. This procedure Verify-Collision was given in [12]; it is represented in Figure 1 for a direct reference. Bins are interpreted in two different ways. When algorithms use a constant number of shared memory registers, then bins are typically interpreted as future rounds during which verification for a collision will be performed, one verification in $\mathcal{O}(1)$

Procedure VERIFY-COLLISION (x)

```
initialize Heads[x] ← Tails[x] ← false
toss_v ← outcome of tossing a fair coin
if toss_v = tails
   then Tails[x] ← true
   else Heads[x] ← true
return Tails[x] = Heads[x]
```

Figure 1 A pseudocode for a processor v of a Common PRAM, where x is a positive integer. `Heads` and `Tails` are arrays of shared memory cells. When the parameter x is dropped in a call then this means that $x = 1$. The procedure returns `true` when a collision has been detected.

rounds. In such a scenario, procedure VERIFY-COLLISION is used without a parameter, because just one shared memory register is needed to carry out one verification. When algorithms use an unbounded array of shared registers, then bins are typically interpreted as some designated shared registers. In such a scenario, procedure VERIFY-COLLISION is invoked with a parameter indicating which bin is verified for collision, because multiple verifications for collisions in different bins can be performed concurrently.

▶ **Lemma 1** ([12, 13]). *For an integer x, procedure* VERIFY-COLLISION (x) *executed by one processor never detects a collision, and when multiple processors execute this procedure then a collision is detected with probability at least $\frac{1}{2}$.*

Properties of naming algorithms

Randomized naming algorithms are categorized as either Monte Carlo or Las Vegas, which are defined as follows. A randomized algorithm is *Las Vegas* when it terminates almost surely and the algorithm returns a correct output upon termination. A randomized algorithm is *Monte Carlo* when it terminates almost surely and an incorrect output may be produced upon termination, but the probability of error converges to zero with the size of input growing unbounded. The naming algorithms we develop are all Monte Carlo and have the probability of error converging to zero with a rate that is polynomial in n. Moreover, when incorrect names are assigned, then the set of integers used as names makes a contiguous segment starting from the smallest name 1 and the only possible kind of error is that duplicate names are given.

A naming algorithm cannot be Las Vegas when n is unknown, as was observed by Kutten et al. [22] for asynchronous computations against an oblivious adversary. An analogous fact holds for synchronous computations.

▶ **Proposition 1.** *There is no Las Vegas naming algorithm for a PRAM with $n > 1$ processors that does not refer to the number of processors n in its code.*

Proof. Let us suppose, to arrive at a contradiction, that such a naming Las Vegas algorithm exists. Consider a system of $n-1 \geq 1$ processors, and an execution $\mathcal{E}$ on these $n-1$ processors that uses specific strings of random bits such that the algorithm terminates in $\mathcal{E}$ with these random bits. Such strings of random bits exist because the algorithm terminates almost surely.

Let v_1 be a processor that halts latest in $\mathcal{E}$ among the $n-1$ processors. Let $\alpha_{\mathcal{E}}$ be the string of random bits generated by processor v_1 by the time it halts in $\mathcal{E}$. Consider an execution $\mathcal{E}'$ on $n \geq 2$ processors such that n processors obtain the same strings of random bits as in $\mathcal{E}$ and an extra processor v_2 obtains $\alpha_{\mathcal{E}}$ as its random bits. The executions $\mathcal{E}$ and $\mathcal{E}'$ are indistinguishable for the $n-1$ processors participating in $\mathcal{E}$, so they assign themselves the same names and halt. Processor v_2 performs the same reads and writes as processor v_1 and assigns itself the same name as processor v_1 does and halts in the same round as processor v_1. This is the termination round because by that time all the other processor have halted as well.

It follows that execution $\mathcal{E}'$ results in a name being duplicated. The probability of duplication for n processors is at least as large as the probability to generate two identical finite random strings in $\mathcal{E}'$ for some two processors, so this probability is positive. ◀

We give algorithms that use the expected number of $\mathcal{O}(n \log n)$ random bits with large probability. The following fact allows to argue about their optimality with respect to the number of random bits.

▶ **Proposition 2** ([12, 13]). *If a randomized naming PRAM algorithm executed by n anonymous processors is correct with some probability p_n then it requires $\Omega(n \log n)$ random bits with the same probability p_n.*

If n is unknown, then the restriction $\mathcal{O}(n \log n)$ on the number of random bits makes it inevitable that the probability of error is at least polynomially bounded from below, as we show next.

▶ **Proposition 3.** *For unknown n, if a randomized naming algorithm is executed by n anonymous processors, then an execution is incorrect, in that duplicate names are assigned to distinct processors, with probability that is at least $n^{-\Omega(1)}$, assuming that the algorithm uses $\mathcal{O}(n \log n)$ random bits with probability $1 - n^{-\Omega(1)}$.*

Proof. Suppose the algorithm uses at most $cn \lg n$ random bits with probability p_n when executed by a system of n processors, for some constant $c > 0$. Then one of these processors uses at most $c \lg n$ bits with probability p_n, by the pigeonhole principle.

Consider an execution for $n+1$ processors. Let us distinguish a processor v. Consider the actions of the remaining n processors: one of them, say w, uses at most $c \lg n$ bits with the probability p_n. Processor v generates the same string of bits with probability $2^{-c \lg n} = n^{-c}$. The random bits generated by w and v are independent. Therefore duplicate names occur with probability at least $n^{-c} \cdot p_n$. When we have a bound $p_n = 1 - n^{-\Omega(1)}$, then the probability of duplicate names is at least $n^{-c}(1 - n^{-\Omega(1)}) = n^{-\Omega(1)}$. ◀

In gauging the optimality of performance of naming algorithms, we will refer to lower bounds on time of such algorithms that can be found in [12, 13], we restate them here for easy reference.

▶ **Theorem 2** ([12, 13]). *A randomized naming algorithm for a Common PRAM with n processors and $C > 0$ shared memory cells operates in $\Omega(n \log n/C)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

▶ **Theorem 3** ([12, 13]). *A randomized naming algorithm for an Arbitrary PRAM with n processors and $C > 0$ shared memory cells operates in $\Omega(n/C)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

The following fact holds for both Common and Arbitrary PRAMs.

▶ **Theorem 4** ([12, 13]). *A randomized naming algorithm for a PRAM with n processors operates in $\Omega(\log n)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

3 Arbitrary with Bounded Memory

We develop a naming algorithm for an Arbitrary PRAM with a constant number of shared memory cells. The algorithm is called Arbitrary-Bounded-MC and its pseudocode is given in Figure 2.

The underlying idea is to have all processors repeatedly attempt to obtain tentative names and terminate when the probability of duplicate names is gauged to be sufficiently small. To this end, each processor writes an integer selected from a suitable "selection range" into a shared memory register and next reads this register to verify whether the write was successful or not. A successful write results in each such a processor getting a tentative name by reading and incrementing another shared register operating as a counter. One of the challenges here is to determine a selection range from which random integers are chosen for writing. A good selection range is large enough with respect to the number of writers, which is unknown, because when the range is too small then multiple processors may select the same integer and so all of them get the same tentative name after this integer gets written successfully. The algorithm keeps the size of a selection range growing with each failed attempt to assign tentative names.

There is an inherent tradeoff here, since on the one hand, we want to keep the size of used shared memory small, as a measure of efficiency of the algorithm, while, at the same time, the larger the range of memory the smaller the probability of collision of random selections from a selection range and so of the resulting duplicate names. Additionally, increasing the selection range repeatedly costs time for each such a repetition, while we also want to minimize the running time as the metric of performance. The algorithm keeps increasing the selection range with a quadratic rate, which turns out to be sufficient to optimize all the performance metrics we measure. The algorithm terminates when the number of selected integers from the current selection range makes a sufficiently small fraction of the size of the used range.

The structure of the pseudocode in Figure 2 is determined by the main repeat-loop. Each iteration of this loop begins with doubling the variable k, which determines the selection range $[1, 2^k]$. This means that the size of the selection range increases quadratically with consecutive iterations of the main repeat-loop. A processor begins an iteration of the main loop by choosing an integer uniformly at random from the current selection range $[1, 2^k]$. There is an inner repeat-loop, nested within the main loop, which assigns tentative names depending on the random selections just made.

All processors repeatedly write to a shared variable `Pad` and next read to verify if the write was successful. It is possible that different processors attempt to write the same value and then verify that their write was successful. The shared variable `Last-Name` is used to proceed through consecutive integers to provide tentative names to be assigned to the latest successful writers. When multiple processors attempt to write the same value to `Pad` and it gets written successfully, then all of them obtain the same tentative name. The variable `Last-Name`, at the end of each iteration of the inner repeat-loop, equals the number of occupied bins. The shared variable `All-Named` is used to verify if all processors have tentative names. The outer loop terminates when the number of assigned names, which is

Algorithm ARBITRARY-BOUNDED-MC

```
initialize k ← 1                                  /* initial approximation of lg n */
repeat
   initialize Last-Name ← name_v ← 0
   k ← 2k
   bin_v ← random integer in [1, 2^k]             /* throw a ball into a bin */
   repeat
      All-Named ← true
      if name_v = 0 then
         Pad ← bin_v
         if Pad = bin_v then
            Last-Name ← Last-Name + 1
            name_v ← Last-Name
         else
            All-Named ← false
   until All-Named
until Last-Name ≤ 2^(k/β)
```

Figure 2 A pseudocode for a processor v of an Arbitrary PRAM with a constant number of shared memory cells. The variables `Last-Name`, `All-Named` and `Pad` are shared. The private variable `name` stores the acquired name. The constant $\beta > 0$ is a parameter to be determined by analysis.

the same as the number of occupied bins, is smaller than or equal to $2^{k/\beta}$, where $\beta > 0$ is a parameter to be determined in analysis.

▶ **Theorem 5.** *Algorithm* ARBITRARY-BOUNDED-MC *always terminates, for any $\beta > 0$. For each $a > 0$ there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names, works in time at most cn, and uses at most $cn \ln n$ random bits, all this with probability at least $1 - n^{-a}$.*

Algorithm ARBITRARY-BOUNDED-MC is optimal with respect to the following performance measures: the expected time $\mathcal{O}(n)$, by Theorem 3, the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

4 Arbitrary with Unbounded Memory

We develop a naming algorithm for Arbitrary PRAM with an unbounded amount of shared registers. The algorithm is called ARBITRARY-UNBOUNDED-MC and its pseudocode is given in Figure 3.

The underlying idea is to parallelize the process of selection of names applied in Section 3 in algorithm ARBITRARY-BOUNDED-MC so that multiple processes could acquire information in the same round that later would allow them to obtain names. As algorithm ARBITRARY-BOUNDED-MC used shared registers `Pad` and `Last-Name`, the new algorithm uses arrays of shared registers playing similar roles. The values read-off from `Last-Name` cannot be used directly as names, because multiple processors can read the same values, so we need to distinguish between these values to assign names. To this end, we assign ranks

Algorithm ARBITRARY-UNBOUNDED-MC

```
initialize k ← 1                                   /* initial approximation of lg n */
repeat
   initialize All-Named ← true
   initialize position_v ← (0, 0)
   k ← r(k)
   bin_v ← random integer in [1, 2^k/(βk)]    /* choose a bin for the ball */
   label_v ← random integer in [1, 2^{βk}]    /* choose a label for the ball */
   for i ← 1 to βk do
      if position_v = (0, 0) then
         Pad [bin_v] ← label_v
         if Pad [bin_v] = label_v then
            Last-Name [bin_v] ← Last-Name [bin_v] + 1
            position_v ← (bin_v, Last-Name [bin_v])
   if position_v = (0, 0) then
      All-Named ← false
until All-Named
name_v ← the rank of position_v
```

Figure 3 A pseudocode for a processor v of an Arbitrary PRAM, when the number of shared memory cells is unbounded. The variables `Pad` and `Last-Name` are arrays of shared memory cells, the variable `All-Named` is shared as well. The private variable `name` stores the acquired name. The constant $\beta > 0$ and an increasing function $r(k)$ are parameters.

to processors based on their lexicographic ordering by pairs of numbers determined by `Pad` and `Last-Name`.

The pseudocode in Figure 3 is structured as a repeat-loop. In the first iteration, the parameter k equals 1, and in subsequent ones is determined by iterations of the increasing integer-valued function $r(k)$, which is a parameter. We consider two instantiations of the algorithm, determined by $r(k) = k + 1$ and by $r(k) = 2k$. In one iteration of the main repeat-loop, a processor uses two variables $\texttt{bin} \in [1, 2^k/(\beta k)]$ and $\texttt{label} \in [1, 2^{\beta k}]$, which are selected independently and uniformly at random from the respective ranges.

We interpret `bin` as a bin's number and `label` as a label for a ball. Processors write their values `label` into the respective bin by instruction $\texttt{Pad}\,[\texttt{bin}] \leftarrow \texttt{label}$ and verify what value got written. After a successful write, a processor increments $\texttt{Last-Name}[\texttt{bin}]$ and assigns the pair $(\texttt{bin}, \texttt{Last-Name}\,[\texttt{bin}])$ as its *position*. This is repeated βk times by way of iterating the inner for-loop. This loop has a specific upper bound βk on the number of iterations because we want to ascertain that there are at most βk balls in each bin. The main repeat-loop terminates when all values attempted to be written actually get written. Then processors assign themselves names according to the ranks of their positions. The array `Last-Name` is assumed to be initialized to 0's, and in each iteration of the repeat-loop we use a fresh region of shared memory to allocate this array.

▶ **Theorem 6.** *Algorithm* ARBITRARY-UNBOUNDED-MC *always terminates, for any* $\beta > 0$. *For each* $a > 0$, *there exists* $\beta > 0$ *and* $c > 0$ *such that the algorithm assigns unique names and has the following additional properties with probability* $1 - n^{-a}$. *If* $r(k) = k + 1$ *then*

at most $cn/\ln n$ *memory cells are ever needed,* $cn\ln^2 n$ *random bits are ever generated, and the algorithm terminates in time* $\mathcal{O}(\log^2 n)$*. If* $r(k) = 2k$ *then at most* $cn^2/\ln n$ *memory cells are ever needed,* $cn\ln n$ *random bits are ever generated, and the algorithm terminates in time* $\mathcal{O}(\log n)$*.*

The instantiations of algorithm Arbitrary-Unbounded-MC are close to optimality with respect to some of the performance metrics we consider, depending on whether $r(k) = k + 1$ or $r(k) = 2k$. If $r(k) = k + 1$ then the algorithm's use of shared memory would be optimal if its time were $\mathcal{O}(\log n)$, by Theorem 3, but as it is, the algorithm misses space optimality by at most a logarithmic factor, since the algorithm's running time is $\mathcal{O}(\log^2 n)$. Similarly, if $r(k) = k+1$ then the number of random bits ever generated $\mathcal{O}(n\log^2 n)$ misses optimality by at most a logarithmic factor, by Proposition 2. On the pother hand, if $r(k) = 2k$ then the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 4, the expected number of random bits $\mathcal{O}(n\log n)$ is optimal, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$ is optimal, by Proposition 3, but the amount of used shared memory misses optimality by at most a polynomial factor, by Theorem 3.

5 Common with Bounded Memory

Algorithm Common-Bounded-MC, which we present in this section, solves the naming problem for Common PRAM with a constant number of shared read-write registers. The algorithm has its pseudocode in Figure 6. To make the exposition of this algorithm more modular, we use two procedures Estimate-Size and Extend-Names. The pseudocodes of these procedures are given in Figures 4 and 5, respectively. The private variables in the pseudocode in Figure 6 have the following meaning: `size` is an approximation of the number of processors n, and `number-of-bins` determines the size of the range of bins we throw conceptual balls into.

The main task of procedure Estimate-Size is to produce an estimate of the number n of processors. Procedure Extend-Names is iterated multiple times, each iteration is intended to assign names to a group of processors. This is accomplished by the processors selecting integer values at random, interpreted as throwing balls into bins, and verifying for collisions. Each selection of a bin is followed by a collision detection. A ball placement without a detected collision results in a name assigned, otherwise the involved processors try again to throw balls into a range of bins. The effectiveness of the resulting algorithm hinges of calibrating the number of bins to the expected number of balls to be thrown.

Balls into bins for the first time

The role of procedure Estimate-Size, when called by algorithm Common-Bounded-MC, is to estimate the unknown number of processors n, which is returned as `size`, to assign a value to variable `number-of-bins`, and assign values to each private variable `bin`, which indicates the number of a selected bin in the range $[1, \texttt{number-of-bins}]$. The procedure tries consecutive values of k as approximations of $\lg n$. For a given k, an experiment is carried out to throw n balls into $k2^k$ bins. The execution stops when the number of occupied bins is at most 2^k, and then $3 \cdot 2^k$ is treated as an approximation of n and $k2^k$ is the returned number of bins.

▶ **Lemma 7.** *For* $n \geq 20$ *processors, procedure* Estimate-Size *returns an estimate* `size` *of* n *such that the inequality* $\texttt{size} < 6n$ *holds with certainty and the inequality* $n < \texttt{size}$ *holds with probability* $1 - 2^{-\Omega(n)}$*.*

Procedure ESTIMATE-SIZE

```
initialize k ← 2                                   /* initial approximation of lg n */
repeat
   k ← k + 1
   bin_v ← random integer in [1, k 2^k]
   initialize Nonempty-Bins ← 0
   for i ← 1 to k 2^k do
      if bin_v = i then
         Nonempty-Bins ← Nonempty-Bins + 1
until Nonempty-Bins ≤ 2^k
return (3 · 2^k, k 2^k)                  /* 3 · 2^k is size, k 2^k is number-of-bins */
```

Figure 4 A pseudocode for a processor v of a Common PRAM. This procedure is invoked by algorithm COMMON-BOUNDED-MC in Figure 6. The variable `Nonempty-Bins` is shared.

Procedure EXTEND-NAMES's behavior can also be interpreted as throwing balls into bins, where a processor v's ball is in a bin x when $\texttt{bin}_v = x$. The procedure first verifies the suitable range of bins $[1, \texttt{number-of-bins}]$ for collisions. A verification for collisions takes either just a constant time or $\Theta(\log n)$ time.

A constant verification occurs when there is no ball in the considered bin i, which is verified when the line "`if` $\texttt{bin}_x = i$ for some processor x" in the pseudocode in Figure 5 is to be executed. Such a verification is performed by using a shared register initialized to 0, into which all processors v with $\texttt{bin}_v = i$ write 1, then all the processors read this register, and if the outcome of reading is 1 then all write 0 again, which indicates that there is at least one ball in the bin, otherwise there is no ball.

A logarithmic-time verification of collision occurs when there is some ball in the corresponding bin. This triggers calling procedure VERIFY-COLLISION precisely $\beta \lg n$ times; notice that this procedure has the default parameter 1, as only one bin is verified at a time. Ultimately, when a collision is not detected for some processor v whose ball is the bin, then this processor increments `Last-Name` and assigns its new value as a tentative name. Otherwise, when a collision is detected, processor v places its ball in a new bin when the last line in Figure 5 is executed.

To prepare for the next round of throwing balls, the variable `number-of-bins` may be reset. During one iteration of the main repeat-loop of the pseudocode of algorithm COMMON-BOUNDED-MC in Figure 6, the number of bins is first set to a value that is $\Theta(n \log n)$ by procedure ESTIMATE-SIZE. Immediately after that, it is reset to $\Theta(n)$ by the first call of procedure EXTEND-NAMES, in which the instruction $\texttt{number-of-bins} \leftarrow \texttt{size}$ is performed. Here, we need to notice that $\texttt{number-of-bins} = \Theta(n \log n)$ and $\texttt{size} = \Theta(n)$, by the pseudocodes in Figures 4 and 6 and Lemma 7.

In the course of analysis of performance of procedure EXTEND-NAMES, we consider a balls-into-bins process; we call it simply the *ball process*. It proceeds through stages so that in a stage we have a number of balls which we throw into a number of bins. The sets of bins used in different stages are disjoint. The number of balls and bins used in a stage are as determined in the pseudocode in Figure 5, which means that there are n balls and the numbers of bins are as determined by an execution of procedure ESTIMATE-SIZE, that is,

Procedure EXTEND-NAMES

```
initialize Collision-Detected ← collision_v ← false
for i ← 1 to number-of-bins do
   if bin_x = i for some processor x then
     if bin_v = i then
       for j ← 1 to β lg size do
         if VERIFY-COLLISION then
           Collision-Detected ← collision_v ← true
       if not collision_v then
         Last-Name ← Last-Name + 1
         name_v ← Last-Name
         bin_v ← 0
if (number-of-bins > size) then
   number-of-bins ← size
if collision_v then
   bin_v ← random integer in [1, number-of-bins]
```

Figure 5 A pseudocode for a processor v of a Common PRAM. This procedure invokes procedure VERIFY-COLLISION, whose pseudocode is in Figure 1, and is itself invoked by algorithm COMMON-BOUNDED-MC in Figure 6. The variables `Last-Name` and `Collision-Detected` are shared. The private variable `name` stores the acquired name. The constant $\beta > 0$ is to be determined in analysis.

the first stage uses `number-of-bins` bins and subsequent stages use `size` bins, as returned by ESTIMATE-SIZE.

The only difference between the ball process and the actions of procedure EXTEND-NAMES is that collisions are detected with certainty in the ball process rather than being tested for. In particular, the parameter β is not involved in the ball process (nor in its name). The ball process terminates in the first stage in which no multiple bins are produced, so that there are no collisions among the balls.

▶ **Lemma 8.** *The ball process modeling the actions of procedure* EXTEND-NAMES *results in all balls ending single in their bins and the number of times a ball is thrown, summed over all the stages, being $\mathcal{O}(n)$, both events occurring with probability $1 - n^{-\Omega(\log n)}$.*

The following Theorem 9 summarizes the performance of algorithm COMMON-BOUNDED-MC (see the pseudocode in Figure 6) as a Monte Carlo one.

▶ **Theorem 9.** *Algorithm* COMMON-BOUNDED-MC *terminates almost surely. For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names, works in time at most $cn \ln n$, and uses at most $cn \ln n$ random bits, each among these properties holding with probability at least $1 - n^{-a}$.*

Proof. One iteration of the main repeat-loop suffices to assign names with probability $1 - n^{-\Omega(\log n)}$, by Lemma 8. This means that the probability of not terminating by the ith iteration is at most $(n^{-\Omega(\log n)})^i$, which converges to 0 with i growing to infinity.

The algorithm returns duplicate names only when a collision occurs that is not detected by procedure VERIFY-COLLISION. For a given multiple bin, one iteration of this procedure

Algorithm COMMON-BOUNDED-MC

```
repeat
   initialize Last-Name ← 0
   (size, number-of-bins) ← ESTIMATE-SIZE
   for ℓ ← 1 to lg size do
      EXTEND-NAMES
      if not Collision-Detected then return
```

Figure 6 A pseudocode for a processor v of a Common PRAM, where there is a constant number of shared memory cells. Procedures ESTIMATE-SIZE and EXTEND-NAMES have their pseudocodes in Figures 4 and 5, respectively. The variables `Last-Name` and `Collision-Detected` are shared.

does not detect collision with probability at most $1/2$, by Lemma 1. Therefore $\beta \lg \texttt{size}$ iterations do not detect collision with probability $\mathcal{O}(n^{-\beta/2})$, by Lemma 7. The number of nonempty bins ever tested is at most dn, for some constant $d > 0$, by Lemma 8, with the suitably large probability. Applying the union bound results in the estimate n^{-a} on the probability of error for sufficiently large β.

The duration of an iteration of the inner for-loop is either constant, then we call is *short*, or it takes time $\mathcal{O}(\log \texttt{size})$, then we call it *long*. First, we estimate the total time spent on short iterations. This time in the first iteration of the inner for-loop is proportional to `number-of-bins` returned by procedure ESTIMATE-SIZE, which is at most $6n \cdot \lg(6n)$, by Lemma 7. Each of the subsequent iterations takes time proportional to `size`, which is at most $6n$, again by Lemma 7. We obtain that the total number of short iterations is $\mathcal{O}(n \log n)$ in the worst case. Next, we estimate the total time spent on long iterations. One such an iteration has time proportional to $\lg \texttt{size}$, which is at most $\lg 6n$ with certainty. The number of such iterations is at most dn with probability $1 - n^{-\Omega(\log n)}$, for some constant $d > 0$, by Lemma 8. We obtain that the total number of long iterations is $\mathcal{O}(n \log n)$, with the correspondingly large probability. Combining the estimates for short and long iterations, we obtain $\mathcal{O}(n \log n)$ as a bound on time of one iteration of the main repeat-loop. One such an iteration suffices with probability $1 - n^{-\Omega(\log n)}$, by Lemma 8.

Throwing one ball uses $\mathcal{O}(\log n)$ random bits, by Lemma 7. The number of throws is $\mathcal{O}(n)$ with the suitably large probability, by Lemma 8. ◀

Algorithm COMMON-BOUNDED-MC is optimal with respect to the following performance metrics: the expected time $\mathcal{O}(n \log n)$, by Theorem 2, the number of random bits $\mathcal{O}(n \log n)$, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

6 Common with Unbounded Memory

We consider naming on a Common PRAM in the case when the amount of shared memory is unbounded. The algorithm we propose, called COMMON-UNBOUNDED-MC, is similar to algorithm COMMON-BOUNDED-MC in Section 5, in that it involves a randomized experiment to estimate the number of processors of the PRAM. Such an experiment is then followed by repeatedly throwing balls into bins, testing for collisions, and throwing again if a collision is detected, until eventually no collisions are detected.

Procedure GAUGE-SIZE-MC

```
k ← 1
repeat
    k ← r(k)
    bin_v ← random integer in [1, 2^k]
until the number of selected values of variable bin is ≤ 2^k/β
return ( ⌈2^{k+1}/β⌉ )
```

Figure 7 A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant $\beta > 0$ is the same parameter as in Figure 8, and an increasing function $r(k)$ is also a parameter.

Algorithm COMMON-UNBOUNDED-MC has its pseudocode given in Figure 8. The algorithm is structured as a repeat loop. An iteration starts by invoking procedure GAUGE-SIZE, whose pseudocode is in Figure 7. This procedure returns `size` as an estimate of the number of processors n. Next, a processor chooses randomly a bin in the range $[1, 3\texttt{size}]$. Then it keeps verifying for collisions $\beta \lg \texttt{size}$, in such a manner that when a collision is detected then a new bin is selected form the same range. After such $\beta \lg \texttt{size}$ verifications and possible new selections of bins, another $\beta \lg \texttt{size}$ verifications follow, but without changing the selected bins. When no collision is detected in the second segment of $\beta \lg \texttt{size}$ verifications, then this terminates the repeat-loop, which triggers assigning each station the rank of the selected bin, by a prefix-like computation. If a collision is detected in the second segment of $\beta \lg \texttt{size}$ verifications, then this starts another iteration of the main repeat-loop.

Procedure GAUGE-SIZE-MC returns an estimate of the number n of processors in the form 2^k, for some positive integer k. It operates by trying various values of k, and, for a considered k, by throwing n balls into 2^k bins and next counting how many bins contain balls. Such counting is performed by a prefix-like computation, whose pseudocode is omitted in Figure 7. The additional parameter $\beta > 0$ is a number that affects the probability of underestimating n.

The way in which selections of numbers k is performed is controlled by function $r(k)$, which is a parameter. We will consider two instantiations of this function: one is function $r(k) = k + 1$ and the other is function $r(k) = 2k$.

▶ **Lemma 10.** *If $r(k) = k + 1$ then the value of `size` as returned by* GAUGE-SIZE-MC *satisfies $\texttt{size} \leq 2n$ with certainty and the inequality $\texttt{size} \geq n$ holds with probability $1 - \beta^{-n/3}$.*

If $r(k) = 2k$ then the value of `size` as returned by GAUGE-SIZE-MC *satisfies $\texttt{size} \leq 2\beta n^2$ with certainty and $\texttt{size} \geq \beta n^2/2$ with probability $1 - \beta^{-n/3}$.*

The following Theorem 11 summarizes the performance of algorithm COMMON-UNBOUNDED-MC (see the pseudocode in Figure 8) as a Monte Carlo one. Its proof relies on mapping an execution of the β-process with verifications on executions of algorithm COMMON-UNBOUNDED-MC in a natural manner.

▶ **Theorem 11.** *Algorithm* COMMON-UNBOUNDED-MC *terminates almost surely, for a sufficiently large β. For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names and has the following additional properties with probability $1 - n^{-a}$.*

Algorithm COMMON-UNBOUNDED-MC

```
repeat
   size ← GAUGE-SIZE
   bin_v ← random integer in [1, 3 size]
   for i ← 1 to β lg size do
      if VERIFY-COLLISION (bin_v) then
         bin_v ← random number in [1, 3 size]
   Collision-Detected ← false
   for i ← 1 to β lg size do
      if VERIFY-COLLISION (bin_v) then
         Collision-Detected ← true
until not Collision-Detected
name_v ← the rank of bin_v among selected bins
```

Figure 8 A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant $\beta > 0$ is a parameter impacting the probability of error. The private variable `name` stores the acquired name.

If $r(k) = k + 1$ then at most cn memory cells are ever needed, $cn \ln^2 n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log^2 n)$. If $r(k) = 2k$ then at most cn^2 memory cells are ever needed, $cn \ln n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log n)$.

The instantiations of algorithm COMMON-UNBOUNDED-MC are close to optimality with respect to some of the performance metrics we consider, depending on whether $r(k) = k + 1$ or $r(k) = 2k$. If $r(k) = k + 1$ then the algorithm's use of shared memory would be optimal if its time were $\mathcal{O}(\log n)$, by Theorem 3, but it misses space optimality by at most a logarithmic factor, since the algorithm's time is $\mathcal{O}(\log^2 n)$. Similarly, for this case of $r(k) = k + 1$, the number of random bits ever generated $\mathcal{O}(n \log^2 n)$ misses optimality by at most a logarithmic factor, by Proposition 2. In the other case of $r(k) = 2k$, the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 4, the expected number of random bits $\mathcal{O}(n \log n)$ is optimal, by Proposition 2, and the probability of error $n^{-\mathcal{O}(1)}$ is optimal, by Proposition 3, but the amount of used shared memory misses optimality by at most a polynomial factor, by Theorem 4.

7 Conclusion

We considered four variants of the naming problem for an anonymous PRAM, when the number of processors n is unknown, and developed Monte Carlo naming algorithms for each of them. The two algorithms for a bounded number of shared registers are provably optimal with respect to the following three performance metrics: expected time, expected number of generated random bits and probability of error. It is an open problem to develop Monte Carlo algorithms for Arbitrary and Common PRAMs for the case when the amount of shared memory is unbounded, such that they are simultaneously asymptotically optimal with respect to these same three performance metrics: the expected time, the expected number of generated random bits and the probability of error.

References

1. Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM*, 61(3):18:1–18:51, 2014.
2. Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the* 24*th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.
3. Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the* 12*th ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.
4. James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209–219, 2006.
5. James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the* 34*th ACM Symposium on Theory of Computing (STOC)*, pages 524–533, 2002.
6. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
7. Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.
8. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley, 2nd edition, 2004.
9. Paolo Boldi and Sebastiano Vigna. An effective characterization of computability in anonymous networks. In *Proceedings of the* 15*th International Conference on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2001.
10. François Bonnet and Michel Raynal. The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. *ACM Transactions on Autonomous and Adaptive Systems*, 6(4):23, 2011.
11. Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitanyi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3):167–176, 2006.
12. Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Anonymous processors with synchronous shared memory: Las Vegas algorithms. Submitted.
13. Bogdan S. Chlebus, Gianluca De Marco, and Muhammed Talo. Anonymous processors with synchronous shared memory. *CoRR*, abs/1507.02272, 2015.
14. Bogdan S. Chlebus and Dariusz R. Kowalski. Asynchronous exclusive selection. In *Proceedings of the* 27*th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–384, 2008.
15. Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1):19–38, 1994.
16. Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. The MIT Press, 1990.
17. Yuval Emek, Jochen Seidel, and Roger Wattenhofer. Computability in anonymous networks: Revocable vs. irrecovable outputs. In *Proceedings of the* 41*st International Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 183–195. Springer, 2014.
18. Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

19 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

20 Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

21 Prasad Jayanti and Sam Toueg. Wakeup under read/write atomicity. In *Proceedings of the 4th International Workshop on Distributed Algorithms (WDAG)*, volume 486 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 1990.

22 Shay Kutten, Rafail Ostrovsky, and Boaz Patt-Shamir. The Las-Vegas processor identity problem (How and when to be unique). *Journal of Algorithms*, 37(2):468–494, 2000.

23 Richard J Lipton and Arvin Park. The processor identity problem. *Information Processing Letters*, 36(2):91–94, 1990.

24 Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.

25 Eric Ruppert. The anonymous consensus hierarchy and naming problems. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2007.

26 Naoshi Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 173–179, 1999.

Lower Bounds on the Amortized Time Complexity of Shared Objects

Hagit Attiya[*1] **and Arie Fouren**[†2]

1 **Department of Computer Science, Technion, Haifa 32000, Israel**
hagit@cs.technion.ac.il

2 **Faculty of Business Administration, Ono Academic College, Kiryat Ono, 5545173, Israel**
aporan@ono.ac.il

Abstract

The *amortized* step complexity of an implementation measures its performance as a whole, rather than the performance of individual operations. Specifically, the amortized step complexity of an implementation is the average number of steps performed by invoked operations, in the worst case, taken over all possible executions. The amortized step complexity of a wide range of known lock-free implementations for shared data structures, like stacks, queues, linked lists, doubly-linked lists and binary trees, includes an additive factor linear in the *point contention*—the number of processes simultaneously active in the execution.

This paper shows that an additive factor, linear in the point contention, is inherent in the amortized step complexity for lock-free implementations of many distributed data structures, including stacks, queues, heaps, linked lists and search trees.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases monotone objects, stacks and queues, trees, step complexity, remote memory references

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.16

1 Introduction

Evaluating the complexity of lock-free implementations, in which an operation may never terminate, is best done through their *amortized* step complexity, defined as the average number of steps performed by invoked operations, in the worst case taken over all possible executions [12]. Amortized step complexity measures the performance of the system as a whole, rather than the performance of individual operations.

Ruppert [12] defined this complexity measure and observed that upper bounds on the amortized step complexity of a wide range of lock-free implementations of shared data structures has an additive factor of $\dot{c}(op)$; $\dot{c}(op)$ is the *point contention* during an operation op, namely, the number of processes simultaneously active during the execution interval of op. These objects include stacks and queues [14], linked lists [6], doubly-linked lists [13] and binary search trees [5]. Ruppert asks whether the additive factor of $\dot{c}(op)$ in the expression for amortized step complexity for lock-free distributed data structures is inherent.

* This work is supported by the Israel Science Foundation (grant number 1749/14)
† This work is supported by the Ono Academic College Research Fund

This paper answers this question in the affirmative, for a wide range of shared data structures, in particular, stacks, queues, heaps, linked lists and search trees. We prove two classes of lower bounds.

The first, bounds the amortized number of *Remote Memory References* (*RMR*s) and it is done by reduction to a variant of the set object, called sack. A sack supports two operations: one operation adds an element, while another atomically removes an element and returns it (returning $\perp$ if the sack is empty). We prove that the amortized RMR complexity of any implementation of a sack using reads, writes and conditional primitives (such as CAS), is at least $\Omega(\dot{c})$. The proof is by reduction to the lower bound for *mutual exclusion* [10], which we extend to hold for amortized step complexity. We show that several shared objects (i.e., stacks, queues and heaps) can easily be used to implement a sack with an $O(1)$ additional cost. This proves that the amortized RMR complexity of these shared objects is also at least $\Omega(\dot{c})$, when the point contention is in $O(\sqrt{\log\log n})$.

We prove another set of lower bounds on the amortized step complexity of *monotone* objects that do not support an atomic remove operation. The proof holds for data structures that are implemented by a connected graph of nodes, like linked lists, skip lists or search trees. The proof is more self-contained, but it bounds only the *step* complexity—a measure that is larger than the RMR complexity. The lower bound of $\Omega(\dot{c})$ for the amortized step complexity holds for implementations using 1-*revealing primitives*, a class including reads, writes, LL/SC, test&set and CAS [3].[1] The lower bound holds when the point contention is in $O(\log\log n)$.

Ruppert [12] provides analysis showing that the known lock-free implementations of stacks and queues [14] have $O(\dot{c}(op))$ amortized step complexity. In addition, he also observes that:

- The search, put and delete operations of the linked-list implementation presented in [6], have $O(n(op) + \dot{c}(op))$ amortized step complexity, where $n(op)$ is the number of the elements in the list when operation op is invoked.
- The amortized step complexity of the search, put and delete on a non-blocking binary tree [5] is $O(h(op) + \dot{c}(op))$, where $h(op)$ is the height of the tree at the beginning of the operation op.
- The amortized step complexity of put and delete in a doubly-linked list [13] is $O(\dot{c}(op))$.
- The wait-free union-find implementation [2] has $O(\alpha(n)+\dot{c}(op))$ amortized step complexity, where n is the number of elements in the sets and $\alpha(n)$ is the inverse of Ackermann function.

Our lower bounds show that the $\dot{c}(op)$ component in the amortized step complexity of these implementations is inherent, except perhaps for the last one, the wait-free union-find, which remains an interesting open problem.

2 The Computation Model

In the *asynchronous shared-memory* model [10], n *processes*, $p_0, \ldots, p_{n-1}$ communicate by applying *primitive operations* (in short, *primitives*) to shared memory registers. Initially, all shared registers hold the value $\perp$. A process is described as a state machine, with a set of (possibly infinitely many) *states*, one of which is a designated *initial state*, and a state transition function.

[1] Using primitives with more revealing power one can get a more efficient implementation of monotone objects. For example, using swap primitive that is $m/2$-revealing [3], it is possible to implement an add operation for a monotone linked list with $O(1)$ step complexity.

The executions of the system are sequences of events. In each event, based on its current state, a process applies a primitive to a shared memory register and then changes its state, according to the state transition function. An *event* ϕ in which a process p applies a primitive op to register R is denoted by a triple $\langle p, R, op\rangle$. An *execution* α is a (finite or infinite) sequence of events $\phi_0, \phi_1, \phi_2, \ldots$. There are no constraints on the interleaving of events by different processes, reflecting the assumption that processes are asynchronous. The value of variable v after α is denoted $val(v, \alpha)$. Without loss of generality, we assume that the value of a shared register is never set to $\perp$ during an execution.

For an execution α and a set of processes P, $\alpha|_P$ is the sequence of all events in α by processes in P; $\alpha|_{\overline{P}}$ is the sequence of all events in α that are *not* by processes in P. If $P = \{p\}$, we write $\alpha|_p$ instead of $\alpha|_{\{p\}}$ and $\alpha|_{\overline{p}}$ instead of $\alpha|_{\overline{\{p\}}}$. An execution α is *P-only* if $\alpha = \alpha|_P$, and it is *P-free* execution if $\alpha = \alpha|_{\overline{P}}$.

The basic primitives are read and write: A read(R) primitive returns the current value of R and does not change its value. A write(v, R) operation sets the value of R to v, and does not return a value. Every process can read from or write to every register, i.e., registers are *multi-writer multi-reader*. A compare&swap (or CAS for short) primitive works as follows. If the register R holds the value v, then after CAS (R, v, u) the state of R is changed to u and true is returned (the CAS *succeeds*). Otherwise, the state of R remains unchanged and false is returned (the CAS *fails*).

An *implementation* of a high-level object provides algorithms for each high-level operation supported by the object. Some transitions are *requests*, invoking a high-level operation, or *responses* to a high-level operation. When a high-level operation is invoked, the process executes the algorithm associated with the operation, applying primitives to the shared registers, until a response is returned.

Let α' be a finite prefix of an execution α. Process p_i performing a high-level operation op is *active* at the end of α', if α' includes an invocation of op without a return from op. The set of the processes active at the end of α' is denoted $active(\alpha')$. The *point contention* at the end of α', denoted $\dot{c}(\alpha')$, is $|active(\alpha')|$.

Consider an execution α of an algorithm A implementing a high-level operation op. For process p_i executing operation op_i, $step(A, \alpha, op_i)$ is the number events by p_i, when executing op_i in α. The step complexity of A in α, denoted $step(A, \alpha)$, is the maximum of $step(A, \alpha, op_i)$ over all operations op_i of all processes p_i.

The *amortized* step complexity of A in an execution α is the total number of shared-memory events performed by all the operations initiated in α (denoted $initiated(\alpha)$) divided by the number of invoked operations:

$$amortizedStep(A, \alpha) = \frac{\sum_{op_i \in initiated(\alpha)} step(A, \alpha, op_i)}{|initiated(\alpha)|}$$

The amortized step complexity of A is the maximum of $amortizedStep(A, \alpha)$ over all possible executions α of A:

$$amortizedStep(A) = \max_{\alpha}\{amortizedStep(A, \alpha)\}$$

Consider a bounded function $S : \mathcal{N} \mapsto \mathcal{N}$. The step complexity of an algorithm implementing operation op is S-*adaptive to point contention* if for every execution α and every operation op_i with interval β_i, $step(A, \alpha, op_i) \leq S(\dot{c}(\beta_i))$. That is, the step complexity of an operation op_i with interval β_i is bounded by a function of the point contention during β_i.

Similarly, the amortized step complexity of an algorithm A is S-*adaptive to point contention* if for every execution α, $amortizedStep(A, \alpha) \leq S(\dot{c}(\alpha))$. That is, the amortized step complexity of algorithm A in an execution α is bounded by a function of the point contention during α.

In this paper, we consider only the *cache-coherent* (CC) model. In the CC model, each process has a local cache in addition to shared memory. A shared variable accessed by a process is loaded to its local cache and remains locally accessible to the process until it is modified by another process. Such a modification results with an update or an invalidation of the cached variable, according to some *cache-consistency* protocol.

In many mutual-exclusion algorithms, a process busy-waits by repeatedly testing one or more local "spin variables". For such algorithms the number of shared memory operations may be unbounded. Instead of counting the shared-memory operations, we count the number of *remote-memory-references* (RMR) generated by the algorithm [1]. The RMR complexity counts only events that cause process-memory interconnection traffic, and ignores events in busy-wait loops on unchanged local variables.

Consider an execution prefix $\alpha'\phi$ of α, where $\phi_i = \langle p_i, v, op_i \rangle$. Following [10], ϕ_i is an *RMR event* if ϕ_i is the first event in α in which p_i accesses v (that is, p_i does not access v in α') or ϕ_i is the first event in α in which p_i accesses v after v was modified by another process (that is, the last event in α' that modifies v is performed by a process $p_j \neq p_i$).

The definition of *RMR complexity* is the same as the definition of step complexity, except that we count only RMR events. Similarly, the definition of *amortized RMR complexity* is the same as the definition of amortized step complexity, counting only RMR events.

3 Lower Bound for Sack Implementation and Related Objects

In this section we define a variation of set object, called sack, and use it to implement mutual exclusion with additional $O(1)$ cost (Algorithm 1). Then we show that sack can be implemented from queues, stacks and heaps with additional $O(1)$ cost (Lemma 7). An $\Omega(\dot{c})$ lower bound for all these data structures then follows from the $\Omega(\dot{c})$ lower bound for mutual exclusion [10], which we extend to amortized RMR complexity (Appendix A).

A sack object supports the put and draw operations, with the following specification:

S.put(v) adds element v to the sack S

S.draw() removes an arbitrary element from the sack S, and returns the removed element. Note that we do not specify what element should be removed. If before the invocation of the operation the sack is empty, the operation returns $\perp$.

The mutual exclusion algorithm shown in this section uses a sack object combined with a few additional CAS and R/W variables. A process can enter the critical section either through a *fast lane*, winning the mutual exclusion associated with a CAS variable *fastLane*, or through a *waiting room*, implemented using a sack object *waitingRoom*. After passing the fast lane or the waiting room, a process has to win a 2-process mutual exclusion implemented with a CAS variable *gate*, in order to enter the actual critical section (see Figure 1).

In more detail (see Algorithm 1 in the appendix), to enter the critical section, process p_i sets $wants[id_i]$ to TRUE, and adds itself to the set of the waiting processes (Line 1.3). Then it tries to win the fast lane, performing a CAS on the *fastLane* variable (Line 1.4). If the CAS succeeds and p_i successfully writes $\langle \text{OCCUPIED}, id_i \rangle$ to *fastLane*, then p_i accesses the *gate* mutual exclusion from the fast lane side (Line 1.9). If CAS on *fastLane* fails (Line 1.4), then p_i busy waits until $privateGate[id_i]$ becomes OPEN (Line 1.7), and then it accesses the *gate* mutual exclusion from the waiting room side (Line 1.9).

In the exit section, p_i sets its variable $wants[id_i]$ to FALSE (Line 1.12) to indicate to other processes that it is not interested to enter the critical section and thus does need help. To complete clean-up, p_i sets $privateGate[id_i]$ to CLOSED (Line 1.13). If p_i previously entered *gate* mutual exclusion through the fast line (in this case $fastLane = \langle \text{OCCUPIED}, id \rangle$), then it resets *fastLane* to OPEN performing a CAS (Line 1.14).

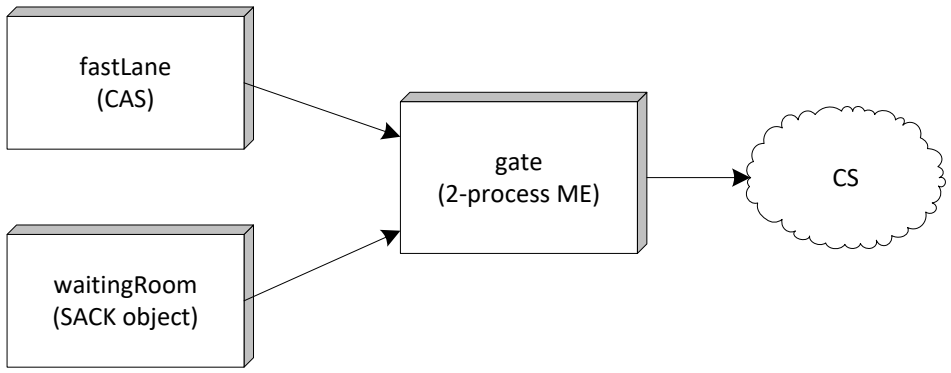


Figure 1 Mutual exclusion using a sack object and CAS presented in Algorithm 1.

Then p_i tries to promote one of the processes from *waitingRoom* to enter the critical section. The R/W variable *promoted* is used to keep track of the promoted process. If in the exit section p_i finds *promoted* equal to its own id_i, then p_i resets *promoted* to $\perp$ (Line 1.16). If p_i finds no promoted process ($promoted = \perp$, Line 1.18), then p_i repeatedly removes a process from *waitingRoom*, until it finds a process *next* that is still interested to enter the critical section, or until *waitingRoom* is empty (Line 1.20). If the removed process *next* is still interested to enter the critical section ($wants[next] =$ TRUE), then p_i promotes this process by setting $promoted = next$ (Line 1.22), and opens the *next*'s private gate by setting $privateGate[next] =$ OPEN (Line 1.23). Finally, p_i releases the critical section associated with *gate* by setting *gate* to TRUE (Line 1.26).

Below we prove the correctness and bound the RMR complexity of Algorithm 1.

The *mutual exclusion* property of the algorithm follows from the mutual exclusion property of the gate block (Figure 1). A process p_i enters critical section only after successful CAS that sets *gate* to CLOSED (Line 1.9) and resets *gate* to OPEN, when it releases the critical section (Line 1.26). Therefore, no two processes may be simultaneously inside the critical section.

The following definition of a *promoted* process is used in the proof of deadlock freedom and in the analysis of the RMR complexity of the algorithm.

A process p_i is *promoted* if $privateGate[id_i] =$ OPEN. The next lemma shows that at most one process is promoted at any point.

▶ **Lemma 1.** *At any point of execution, if* $privateGate[id_i] =$ OPEN, *then* $promoted = id_i$.

Proof. Initially, for every process p_j, $privateGate[id_j] =$ CLOSED and the lemma trivially holds.

Suppose that the lemma holds for an execution prefix α', and let p_i be the first process that modifies *privateGate* after α'. Before p_i sets $privateGate[next] =$ OPEN in its exit section (Line 1.23), p_i sets $promoted = next$ (Line 1.22). By the mutual exclusion property of the algorithm, these steps are performed by p_i in exclusion, and the lemma holds. ◀

The following technical lemma is used to prove that the algorithm has no deadlocks.

▶ **Lemma 2.** *Let* $exit_i$ *be the execution interval corresponding to the exit section of a process* p_i. *Then one of the following holds:*

(a) *at the end of $exit_i$ there is a promoted process p_j, or*
(b) *p_i's invocation of waitingRoom.*draw() *returns $\perp$ (Line 1.19).*

Proof. If p_i reads $promoted = id_j \neq \perp$ (Line 1.18), then by Lemma 1 and the mutual exclusion property, $privateGate[id_j] =$ OPEN at this point, and p_j remains to be promoted until the end of $exit_i$. In this case, condition (a) holds.

If p_i reads $promoted = \perp$ (Line 1.18), and succeeds to set $privateGate[next] =$ OPEN (Line 1.23) then process *next* is promoted at this point, and it remains promoted until the end of $exit_i$, and condition (a) holds.

If p_i reads $promoted = \perp$ (Line 1.18) but it does not perform $privateGate[next] =$ OPEN (Line 1.23), then *waitingRoom.*draw() returns $\perp$ (Line 1.19), implying condition (b). ◀

The next lemma proves the deadlock-freedom property of the algorithm. For simplicity, we assume that each process enters the critical section at most once. Since the lower bound presented in [10] holds for one-shot mutual exclusion, this suffices for our proof.

▶ **Lemma 3.** *Algoritm 1 is deadlock-free.*

Proof. Suppose that there is an execution α with prefix α', such that after α' there is a process p_i in its entry section, and from that point on no process ever enters the critical section in α.

If p_i's CAS on *fastLane* succeeds (Line 1.4), then p_i waits on the *gate* variable (Line 1.9). This is a contradiction, since no process remains in the critical section associated with *gate* forever.

Therefore, p_i loses the *fastLane* in Line 1.4 and remains in *waitingRoom* forever, waiting for $privateGate[id]$ to be OPEN (Line 1.7).

Since p_i's CAS on *fastLane* fails, another process p_j wins *fastLane* in its enter section (Line 1.4) before p_i's CAS, and releases *fastLane* in its exit section (Line 1.14) after p_i's CAS. Process p_j performs *waitingRoom.*draw() (Line 1.19) after it releases *fastLane* (Line 1.14). By assumption, p_i remains in *waitingRoom* forever, and therefore, the *waitingRoom.*draw() invocation of p_j (Line 1.19) does not return $\perp$.

Therefore, statement (a) of Lemma 2 holds, implying that some process p_k is promoted at the end of p_j's exit interval. Eventually, p_k accesses the *gate* mutual exclusion after it reads $privateGate[k] =$ OPEN (Line 1.23) or wins the *fastLane* (Line 1.4). Since *gate* mutual exclusion has no deadlocks, eventually some process enters the critical section after α', which is a contradiction. ◀

Now we show that at any point, at most one process can access the *gate* mutual exclusion from the fast lane, and at most one process from the waiting room side (Figure 1). This implies that the *gate* mutual exclusion has constant RMR complexity (in addition to the RMR complexity of the sack implementation for *waitingRoom*).

▶ **Definition 4.** A process p_i is on *fast lane* if and only if $fastLane = \langle$OCCUPIED$, id_i\rangle$.[2]

By the semantics of CAS and induction on the execution order, we have the following lemma:

▶ **Lemma 5.** *The execution intervals in which different processes p_i and p_j are on fast lane are disjoint.*

[2] That is, p_i is on fast lane after it successfully sets *fastLane* to $\langle$OCCUPIED$, id_i\rangle$ (Line 1.4) and before it successfully resets *fastLane* to EMPTY (Line 1.14) in the exit section.

Lemma 5 implies that at any point, at most one process accesses *gate* mutual exclusion from the fast lane, and Lemma 1 implies that at most one process accesses the *gate* mutual exclusion from the waiting room. We have that at most two processes simultaneously access the *gate* mutual exclusion and therefore, waiting for *gate* (Line 1.9) has a constant amortized RMR complexity (each event that generates RMR can be charged to a successful entry to the *gate* critical section). Each process is added and removed from the waiting room sack at most once, and the rest of the operations in the entry and exit sections have a constant RMR complexity. This implies:

▶ **Lemma 6.** *Given an implementation of* sack *with* $O(f(\dot{c}))$ *amortized RMR complexity. Then there is a mutual exclusion algorithm with* $O(f(\dot{c}))$ *amortized RMR complexity, using in addition,* R/W *and* CAS*.*

The following lemma shows that sack can be implemented from several shared objects, with a constant additional cost:

▶ **Lemma 7.** *The following data structures: queues, stacks and heaps, can be used to implement the* sack *object operations with* $O(1)$ *additional steps.*

Proof. (a) queue operations enqueue and dequeue trivially implement the put and draw operations of sack; (b) stack operations push and pop trivially implement the put and draw operations of sack; (c) heap operations add and removeMax trivially implement the put and draw operations of sack. ◀

Theorem 20 (presented in Appendix A), Lemma 6 and Lemma 7 imply the next theorem:

▶ **Theorem 8.** *Any implementation of queues, stacks and heaps, from reads, writes and conditional primitives has at least* $\Omega(\dot{c})$ *amortized RMR complexity.*

4 Lower Bounds for Graph-Based Set Implementations

This section defines *graph-based* implementations of the set object (denoted graph-based-set), and proves that any implementation of graph-based-set from 1-revealing primitives has an $\Omega(\dot{c})$ amortized step complexity. This is a generalization of data structures that can be represented by a connected graph of nodes, like linked lists, skip lists or search trees. The lower bound on the amortized step complexity of graph-based-set holds also for implementation of these data structures.

To emphasize the nature of the data structures based on a connected set of nodes, this section assumes the following memory model. Each shared variable contains one data structure node, which has a data field and a constant number d of pointers connecting it to other nodes (Figure 2). A process can read or modify all the components of a node in a single computational step.[3]

The state of the shared memory after an execution prefix α can be represented by a *memory graph* $G(\alpha) = (V, E)$, where V is set of the shared variables in the system, and there is an edge $v_i \to v_j \in E$ if and only if after α, the shared variable corresponding to $v_i \in V$ contains a pointer to the shared variable corresponding to $v_j \in V$.

[3] Many known implementations of linked data structures use a separate variable for each node component, and a fixed memory offset to access different components of the node. These implementations can be converted to graph-based-set preserving the asymptotic step complexity, by replacing each data structure node with a linked list of graph-based-set nodes, having a separate node for each component.

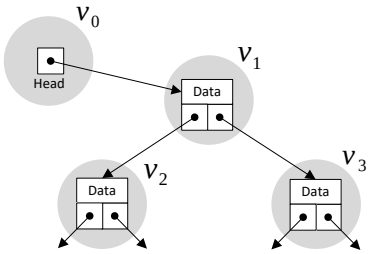


Figure 2 A graph-based representation of a set. Each node of the graph is stored in a separate shared variable v_i and contains a data field and a constant number of pointers. The pointers between the nodes form the edges of the graph. An element v_i is in the set iff it can be traced following the pointers starting from the *head* node.

A graph-based-set implementation specifies a special node, *head*, and supports one operation, add(e), which adds an element e to the set. It should satisfy the following invariant:

- an element e is in the graph-based-set S after an execution prefix α if and only if there is a directed path from $S.head$ to a node v with $v.data = e$, in the memory graph G_α.

Many linked-list and tree-based set implementations [4, 7–9, 11, 16] satisfy this invariant. In contrast to the RMR lower bound of Section 3, the lower bound for graph-based-set only requires an add operation, and does not rely on an atomic remove operation.

We construct an execution of $3k$ processes, in which each process p_i invokes add(id_i) to add its *id* to the set. We show that these $3k$ processes collectively perform $\Omega(k^2)$ steps, implying that any implementation of graph-based-set has at least $\Omega(k) = \Omega(\dot{c})$ amortized step complexity. The proof uses a technique similar to the $\Omega(\dot{c})$ lower bound on the step complexity of adaptive collect [3].

We construct the execution in rounds, maintaining two disjoint sets of processes, the *invisible* set P, initially containing all processes, and the *visible* set W, initially empty. Intuitively, invisible processes are not aware of each other and do not detect each other's operations, while visible processes are possibly aware of other processes or vice versa.

In round r, each process that is still invisible after the previous round performs its next event. These steps are scheduled so that after each round, at most 2 invisible processes become visible. These processes cannot be erased from the execution. Instead, they are stopped and take no steps in the later rounds. Some of the other invisible processes are retroactively erased from the execution in order to keep the remaining processes invisible after round r.

In each round, at most one process succeeds to add its id to the graph-based-set, and at least k processes are invisible after k rounds. Since in each round each invisible process takes one step, in k rounds the last k invisible processes collectively perform $\Omega(k^2)$ steps, implying an $\Omega(\dot{c})$ lower bound on the amortized step complexity.

To guarantee that a process p_i remains invisible during the execution, we need to show that no other invisible process reads information stored by p_i, and that p_i does not modify any variable previously modified by another invisible process. Otherwise, p_i cannot be erased from the execution without affecting other processes. To formalize the notion of variables affected by a process, we define a set of *evidence* variables. Intuitively, the evidence variables of process p_i are the variables that may contain "traces" of p_i's events, so that the values of these variables may change if p_i's events are deleted from the execution.

▶ **Definition 9** (Evidence Set of Variables)**.** The *evidence set* of process p after execution α, denoted $evidence(p, \alpha)$, is the set of the variables v whose values at the end of α would change if the events of p are deleted from α, that is,

$$evidence(p, \alpha) = \{v \mid val(v, \alpha) \neq val(v, \alpha|_{\overline{p}})\}$$

For a set of processes P, $evidence(P, \alpha) = \cup_{p \in P}\ evidence(p, \alpha)$.

Now we define the notion of a set of processes that are invisible to each other. Intuitively, erasing any subset of an invisible set of processes is undetectable by the remaining invisible processes. To facilitate the inductive construction, we require some additional properties.

▶ **Definition 10** (Invisible Set of Processes)**.** A set P of processes is *invisible in an execution* α if the following hold:

(a) for any subset $Q \subset P$, processes $P \setminus Q$ get the same responses in the executions α and $\alpha|_{\overline{Q}}$;

(b) for any subset $Q \subset P$, and for any process $p \in P \setminus Q$, $evidence(p, \alpha) = evidence(p, \alpha|_{\overline{Q}})$;

(c) for any pair of processes $p, q \in P$, $evidence(p, \alpha) \cap evidence(q, \alpha) = \emptyset$.

The corresponding set of *visible* processes are the active processes in α that are not in P, i.e., $W = active(\alpha) \setminus P$.

Definition 10(a) means that erasing any subset Q of the invisible processes P is undetectable to the remaining invisible processes in $P \setminus Q$. Property (b) implies that erasing any subset Q of invisible processes P does not affect the evidence sets of the remaining invisible processes $P \setminus Q$; it prevents an invisible process $p \in P$ from modifying a variable previously modified by another invisible process $q \in P$, and ensures that erasing p will not make q visible. Property (c) states that the evidence sets of invisible processes are disjoint, so each variable belongs to evidence set of at most one process.

The next lemma (proved in [3]) states that after erasing any subset of an invisible set from an execution, the remaining processes still form an invisible set.

▶ **Lemma 11.** *If a set of processes P is invisible in an execution α, then for every subset $Q \subset P$, the subset $(P \setminus Q)$ is invisible in $\alpha|_{\overline{Q}}$.*

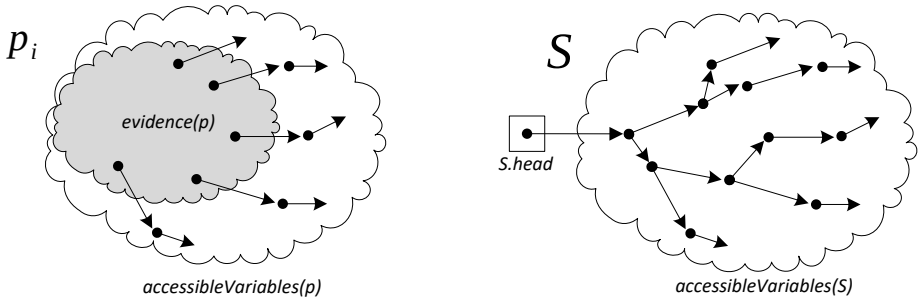
The following definitions of accessible variables and accessible processes are used to bound the size of the underlying graph of graph-based-set.

▶ **Definition 12** (Accessible Variables)**.** A shared variable v is an *accessible variable of a process p_i after an execution prefix α*, if there is a variable $v_0 \in evidence(p_i, \alpha)$, and there is a directed path from v_0 to v in the graph G_α (Figure 3(a)).
The set of the accessible variables of p_i after α is denoted $accessibleVariables(p_i, \alpha)$.

▶ **Definition 13** (Accessible Variables of a graph-based-set)**.** The *accessible variables of a* graph-based-set S *after execution prefix α*, denoted $accessibleVariables(S, \alpha)$, is the set of variables v such that after α there is a directed path from $S.head$ to v in the memory graph G_α (Figure 3(b)).

▶ **Definition 14** (Accessible Processes of a graph-based-set)**.** The *accessible processes of a* graph-based-set S *after execution prefix α*, denoted $accessibleProcesses(S, \alpha)$, are those whose evidence variables are in the set of the accessible variables of S :

$$accessibleProcesses(S, \alpha) = \{p_i : evidence(p_i, \alpha) \cap accessibleVariables(S, \alpha) \neq \emptyset\}$$



(a) (b)

Figure 3 (a) The accessible variables of process p_i are the shared variables that can be accessed through a directed path in the memory graph G_α, starting from a variable in $evidence(p_i, \alpha)$. (b) The accessible variables for a graph-based-set S are the shared variables that can be accessed through a directed path in the graph G_α, starting from $S.head$.

Assume, without loss of generality, that a constant number of primitive types Op_0, $Op_1, \ldots, Op_{t-1}$ is used, and each process applies primitives cyclically in this order during its execution. That is, in its i-th step the process performs a primitive of type $Op_{i \bmod t}$. Any algorithm may be modified to follow this rule, by introducing dummy steps of the required type. This increases the step complexity of the algorithm by a constant factor t, since the algorithm uses a constant number of primitive types, and does not affect the asymptotic step complexity.

Intuitively, the next definition implies that the events of invisible processes accessing the same variable v with the same primitive Op can be ordered in such a way that at most m of these processes are revealed, while the rest of the processes remain invisible.

▶ **Definition 15** (m-Revealing Primitives [3])**.** Suppose that a set of processes P is invisible in an execution α and consider a variable $v \notin evidence(P, \alpha)$. Suppose that there is a subset $P_k \subseteq P$ of k processes whose next events $\phi_1, \ldots, \phi_k$ apply the same primitive Op to the variable v:

$$\forall p \in P_k : \ next\_event(p, \alpha) = \langle p, v, Op \rangle$$

We say that the primitive Op is *m-revealing*, for $m \geq 0$, if there is a permutation π of the next events $\phi_1, \ldots, \phi_k$ and a subset $P_m \subseteq P_k$ of size $\leq m$, such that $(P \setminus P_m)$ is invisible in $\alpha\pi$.

The next lemma provides the induction step for the lower bound proof, leading to a new invisible set P_{r+1}, a corresponding visible set W_{r+1} and a new state of the graph-based-set S. Our goal is to keep the invisible set P_{r+1} as large as possible, and the graph-based-set S as small as possible, in order to carry out the induction for as long as possible.

▶ **Lemma 16.** *Suppose that there is an execution α_r such that:*

(a) *there is an invisible set P_r of size $|P_r| = m_r \geq 2$ and a corresponding visible set W_r of size $|W_r| = w_r$;*

(b) *each process $p_i \in P_r$ performs exactly r steps in α_r;*

(c) *for every pair of processes $p_i, p_j \in P_r$, $accessibleVariables(p_i, \alpha_r) \cap accessibleVariables(p_j, \alpha_r) = \emptyset$;*

(d) *for each process* $p_i \in P_r$, $|evidence(p_i, \alpha_r)| \leq r$ *and* $|accessibleVariables(p_i, \alpha_r)| \leq r(d+1)$ *(recall that* d *is the number of pointers that can be stored in a shared variable);*

(e) *for the* graph-based-set S, $|accessibleProcesses(S, \alpha_r)| \leq r$ *and* $|accessibleVariables(S, \alpha_r)| = \frac{r(r+1)(d+1)}{2}$.

Then there is an execution α_{r+1} *such that after* α_{r+1}*:*

(a1) *there is an invisible set* P_{r+1} *of size* $m_{r+1} \geq \sqrt{\frac{m_r}{2d+3}}$ *and a corresponding visible set* W_{r+1} *of size* $w_{r+1} \leq w_r + 2$*;*

(b1) *each process* $p_i \in P_{r+1}$ *performs exactly* $r+1$ *steps in* α_{r+1}*;*

(c1) *for every pair of processes* $p_i, p_j \in P_r$, $accessibleVariables(p_i, \alpha_{r+1}) \cap accessibleVariables(p_j, \alpha_{r+1}) = \emptyset$.

(d1) *for every process* $p_i \in P_{r+1}$, $|evidence(p_i, \alpha_{r+1})| \leq r+1$ *and* $|accessibleVariables(p_i, \alpha_{r+1})| \leq (d+1)(r+1)$*;*

(e1) *for the* graph-based-set S, $|accessibleProcesses(S, \alpha_{r+1})| \leq r+1$ *and* $|accessibleVariables(S, \alpha_{r+1})| = \frac{(r+1)(r+2)(d+1)}{2}$.

Proof. We show how to extend α_r with one more round to obtain an execution α_{r+1}, and a set P_{r+1} invisible in α_{r+1}, such that each process in P_{r+1} performs $(r+1)$ steps in α_{r+1} and properties (a1)—(e1) hold.

By the induction hypothesis (e), $|accessibleProcesses(S, \alpha_r)| \leq r$. Hence, at most r processes complete their add(id_i) operations in α_r. Since the rest of the invisible processes $p_i \in (P_r \setminus accessibleProcesses(S, \alpha_r))$ do not complete their add(id_i) operations in α_r, they are poised to execute their next event $\phi_i = next\_event(p_i, \alpha_r)$ in round $r+1$. By assumption and property (b), all the events $\phi_1, \phi_2, \ldots$ of the processes P_r in round $r+1$ apply the same 1-revealing primitive.

Four issues must be addressed in order to keep many processes invisible and S small:

(1) Avoid conflicts between the events of round $r+1$ and the events performed in the previous rounds that may violate the properties of invisible set. Otherwise, if process p_i in round $r+1$ accesses a variable in the evidence set of another process p_j, then p_j cannot be later erased from the execution without affecting the execution of p_i. To simplify the proofs, we require not only the evidence sets of the invisible processes to be disjoint, but also their accessible sets.

(2) Ensure that the *accessibleVariables* of the invisible processes are disjoint (c1). The last two kinds of conflicts are eliminated using inductive hypothesis (c) and applying Turán's Theorem.

(3) Ensure there are no conflicts between events in round $r+1$. These are eliminated by using the properties of 1-revealing primitives.

(4) Keep the size of $accessibleVariables(p_i)$ small, for every invisible processes p_i, in order to keep the size of $accessibleVariables(S)$ and $accessibleProcesses(S)$ small.

(1) Eliminating conflicts with the previous rounds. Consider a *visibility* graph $G(V, E)$, with vertices V corresponding to the processes in P_r. There are two kinds of edges in G (see Figure 4): A *solid* edge $p_i \to p_j \in E$ exists if process p_i accesses a variable $v_j \in evidence(p_j, \alpha_r)$, $p_i \neq p_j$, in round $r+1$. A *dashed* edge $p_i \dashrightarrow p_j \in E$ exists if a process p_i writes to a variable v_i a *pointer* to a variable $v_j \in accessibleVariables(p_j, \alpha_r)$, $p_i \neq p_j$, in round $r+1$.

We prove that for each process p_i, there is at most one outgoing solid edge in the graph $G_{\alpha_{r+1}}$. By Definition 10(c) of the invisible set P_r, the evidence sets of the processes P_r are disjoint. Therefore variable v_j belongs to evidence set of at most one process $p_j \in P_r$. In

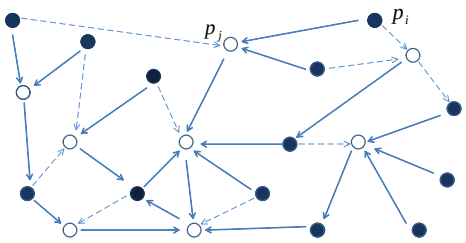


Figure 4 Example of the visibility graph $G(V, E)$ used in the proof of Lemma 16.

round $r+1$, process p_i accesses at most one variable v_j. Therefore, for each process $p_i \in V$ there is at most one outgoing solid edge $p_i \to p_j \in E$.

We also prove that for each process p_i, there are at most d outgoing dashed edges in $G_{\alpha_{r+1}}$. If in round r, process p_i writes to a variable v_i a pointer to a variable $v_j \in$ *accessibleVariables*(p_j, α_r), for some process $p_j \in P_r$. By inductive hypothesis (c), the *accessibleVariables* of invisible processes are disjoint after α_r. Since each variable contains at most d pointers, there are at most d such processes p_j. This implies that for each process p_i, the graph $G_{\alpha_{r+1}}$ contains at most d outgoing dashed edges $p_i \dashrightarrow p_j \in E$.

Therefore, $|E| \leq (d+1)|V|$ and the average degree of $G_{\alpha_{r+1}}$ (as an undirected graph) is

$$2|E|/|V| \leq 2(d+1)|V|/|V| \leq 2(d+1).$$

Next, we apply the Turán's theorem [15]:

▶ **Theorem 17** (Turán). *Let $G(V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. If the average degree of G is r, then $G(V, E)$ has an independent set with at least $\left\lceil \frac{|V|}{r+1} \right\rceil$ vertices.*

It follows that G has an independent set $V' \subseteq V$ with at least $\left\lceil \frac{|V|}{(2d+3)} \right\rceil$ vertices (represented by shaded circles on Figure 4). We leave the processes corresponding to V' in the execution (along with the visible processes W_r that cannot be erased), and erase all the other invisible processes $V \setminus V'$. That is, we define $\alpha'_r = \alpha_r|_{V' \cup W_r}$. By the construction of V', if a process $p \in V'$ is about to access a variable v (in round $r+1$ after α'_r), then this variable is not in the evidence set of any other process in V', and p_i does not write to v a pointer to a variable in the evidence set of another process in V'. Therefore, there are no conflicts between the primitives of round $r+1$ and the primitives of rounds $1, \ldots, r$.

It is still possible that two processes $p, q \in V'$ are about to access the same variable v (not in the evidence set of any one of them) in round $r+1$, violating Properties (a) and (b) of an invisible set (Definition 10). Thus, in order to keep the set of the processes invisible, we should eliminate conflicts between events in round $r+1$. We do this separately for processes that access the variables from *accessibleVariables*(S, α_r), and the processes that do not access graph-based-set S.

(2) Eliminating conflicts between events in the same round. We next order the events of round $r+1$ for the processes that do not access S. Let $M(V')$ be the set of the variables accessed by the processes in V' in round $r+1$. Note that by the construction of V', if distinct processes $p, q \in V'$ access a variable $v \in M(V')$, then v is not in the evidence set of

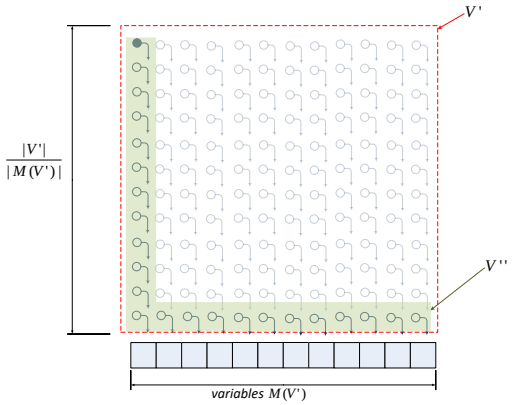


Figure 5 Induction step in the proof of Lemma 16.

any one of them. We use the properties of 1-revealing primitives to keep as many processes in V' invisible as possible, revealing only a small fraction of them. Suppose that after α'_r all processes in V' are about to perform a 1-revealing primitive of type $Op_{(r+1) \bmod t}$.

Let $v_1 \in M(V')$ be the variable that is accessed by the largest number of processes. In round $r+1$, we keep all the processes accessing the variable v_1, and exactly one process for each variable in $M(V') \setminus \{v_1\}$. Let V'' be the set of these processes (Figure 5). We define $\alpha''_r = \alpha'_r|_{V'' \cup W_r}$.

Now we will define the sequence of the next events of processes V'' for round $r+1$. Since $V'' \subseteq P_r$ is invisible in α_r, Lemma 11 implies that V'' is invisible in α''_r. Let $p_{1,1}, p_{1,2}, \ldots, p_{1,l} \in V''$ be the processes that are about to access variable v_1 by performing events $\phi_{1,1}, \phi_{1,2}, \ldots, \phi_{1,l}$. Note that all these events apply the same 1-revealing primitive to v_1. By Definition 15 of a 1-revealing primitive, there is a permutation π_1 of the events $\phi_{1,1}, \phi_{1,2}, \ldots, \phi_{1,l}$ such that after $\alpha''_r\pi_1$, at most one of the processes $p_{1,1}, p_{1,2}, \ldots, p_{1,l}$, say $p(v_1)$, becomes visible while the others remain invisible. By Definition 15, all other processes that are invisible in α''_r are also invisible in $\alpha''_r\pi_1$.

Finally, we schedule the events $\phi_{j_1}, \ldots, \phi_{j_m}$ of the processes that access the variables $M(V') \setminus \{v_1\}$, obtaining the execution $\alpha_{r+1} = \alpha''_r\pi_1\phi_{j_1}, \ldots, \phi_{j_m}$, in which each process in V'' performs exactly $r+1$ steps.

In the execution segment π_1 defined above, only one process, $p(v_1)$, becomes visible, while all other processes remain invisible. Consider the events $\phi_{j_1}, \ldots, \phi_{j_m}$ scheduled at the end of round $r+1$. By the construction, there is exactly one process accessing each of the variables $M(V') \setminus \{v_1\}$, and after $\alpha''_r\pi_1$, none of these variables belongs to evidence set of the processes in V''. Therefore, after $\alpha_{r+1} = \alpha''_r\pi_1\phi_{j_1}, \ldots, \phi_{j_m}$, processes $V'' \setminus \{p(v_1)\}$ form an invisible set, denoted P_{r+1}.

(3) Bounding the size of the invisible set P_{r+1} from below. Since $|V'|$ processes access $|M(V')|$ different variables, on average, $\frac{|V'|}{|M(V')|}$ processes access the same variable. Note that the number of processes that access the variable v_1 is more than the average $\frac{|V'|}{|M(V')|}$. As shown above, all the processes accessing v_1 remain invisible except one process $p(v_i)$.

Therefore, the number of invisible processes after the execution α_{r+1} is

$$|P_{r+1}| = |V''| - 1 \geq \frac{|V'|}{|M(V')|} + |M(V')| - 2 \ .$$

For simplicity of notation, denote $|M(V')| = x$. Using this notation, the number of processes that are invisible after round $r+1$ is $m_{r+1} \geq \frac{|V'|}{x} + x - 2$.

Differentiating by x and equating to 0, we get $m'_{r+1}(x) = -\frac{|V'|}{x^2} + 1 = 0$, implying $x = \sqrt{|V'|}$. Therefore, m_{r+1} is minimized with

$$\frac{|V'|}{\sqrt{|V'|}} + \sqrt{|V'|} - 2 = 2\sqrt{|V'|} - 2 \ .$$

Taking $|V'| = m_r/(2d+3)$ implies $m_{r+1} \geq 2\sqrt{\frac{m_r}{2d+3}} - 2 \geq \sqrt{\frac{m_r}{2d+3}}$, showing property (a1).

(4) Bounding the size of the evidence set and the accessible nodes set. In round $r+1$, each invisible process p_i performs one step at which it accesses at most one variable $v \notin \mathit{evidence}(p_i, \alpha_r)$. Therefore, in round $r+1$, at most one new variable is added to the evidence set of p_i, and the size of the evidence set grows at most by 1, showing that $|\mathit{evidence}(p_i, \alpha_{r+1})| \leq r+1$, as required by the first part of property (d1).

By inductive hypothesis (c), no variable $v_i \in \mathit{accessibleVariables}(p_i, \alpha_r) \setminus \mathit{evidence}(p_i, \alpha_r)$ is in the evidence set of another process p_j. Therefore, no such variable v_i is modified by any process in α_r, it remains in its initial state $\perp$ and does not contain a pointer to another variable. This implies that for every variable $v \in \mathit{evidence}(p_i, \alpha_r)$, there are at most d variables $v_i \in (\mathit{accessibleVariables}(p_i, \alpha_r) \setminus \mathit{evidence}(p_i, \alpha_r))$ accessible from v. This implies

$$|\mathit{accessibleVariables}(p_i, \alpha_r) \setminus \mathit{evidence}(p_i, \alpha_r)| \leq d \cdot |\mathit{evidence}(p_i, \alpha_r)|$$

that is, $|\mathit{accessibleVariables}(p_i, \alpha_r)| \leq (d+1) \cdot |\mathit{evidence}(p_i, \alpha_r)| \leq (d+1)(r+1)$, implying the second part of property (d1).

Finally, we show that the set $\mathit{accessibleVariables}(S)$ can be kept relatively small, while keeping enough processes invisible. Consider the processes $P_{S,r} \subseteq P_r$, whose computational events in round $r+1$ access a variable in $\mathit{accessibleVariables}(S, \alpha_r)$.

Let v be the node of graph-based-set S accessed by the largest number of processes in $P_{S,r}$ in round $r+1$. We keep the processes accessing v (denoting them $P_{S,r+1}$), and erase the rest of the processes $(P_r \setminus P_{S,r+1})$ from the execution. Since all the processes $P_{S,r}$ perform the same 1-revealing primitive in round $r+1$, by Definition 15, there is a permutation of the computation events such that only one of these processes, p_i, becomes visible, and the rest remain invisible.

By induction hypothesis (d), $|\mathit{accessibleVariables}(p_i, \alpha_r)| \leq r(d+1)$. Therefore, in round $r+1$, the size of $\mathit{accessibleProcesses}(S, \alpha_r)$ grows by at most 1, and the size of $\mathit{accessibleVariables}(S, \alpha_r)$ grows by at most $r(d+1)$, implying property (e1) (Figure 6). By inductive hypothesis (e),

$$|\mathit{accessibleVariables}(S, \alpha_r)| \leq \frac{r(r+1)(d+1)}{2}$$

Therefore,

$$|P_{S,r+1}| \geq \frac{|P_S|}{|\mathit{accessibleVariables}(S, \alpha_r)|} \geq \frac{2|P_S|}{r(r+1)(d+1)}$$

The number of processes accessing S decreases more slowly than the number of other invisible processes. Property (a1) bounds from below the number of invisible processes after α_{r+1}. ◀

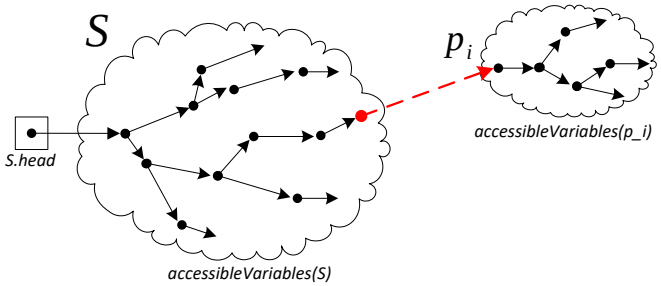


Figure 6 Bounding the size of graph-based-set in Lemma 16.

By Lemma 16, the number of invisible processes after round $r+1$ is $m_{r+1} \geq \sqrt{\frac{m_r}{2d+3}}$.

To prove the lower bound on the amortized step complexity, we start with the empty execution α_0 that satisfies the inductive hypothesis of Lemma 16 with the invisible set P_0 containing all the n processes in the system. Then, we inductively construct execution α_r containing r rounds, using Lemma 16. We require that after α_r there are $|P_r| = r$ invisible processes, i.e., $m_r = r$. Solving this recurrence, we get $m_0 = (2d+3)^{2^r-1} r^{2^r} = \frac{((2d+3)r)^{2^r}}{2d+3}$. Since d is a constant, the total number of the processes in the system should be $n \geq m_0 = \Omega(((2d+3)r)^{2^r})$, or $r = O(\log\log n)$.

By Lemma 16(a1), in each of r rounds of the execution α_r, at most 2 process becomes visible and stopped, and r processes remain invisible after α_r. The rest of the processes are erased from the execution. Therefore, $\dot{c}(\alpha_r) = |P_r| + |W_r| \leq 3r = O(\log\log n)$.

By Lemma 16(e1), at most one process $p_i \in P_r$ completes its operation add(id_i) in round r, and each invisible process takes one step in each round, until it becomes visible. Therefore, the total number of steps performed by the processes P_r in α_r is at least $r|P_r| = r^2 = \Omega(\dot{c}^2)$.

We have constructed an execution of $\dot{c}$ processes that collectively take $\Omega(\dot{c}^2)$ steps. Therefore, the amortized step complexity of an operation add is in $\Omega(\dot{c}^2/\dot{c}) = \Omega(\dot{c})$, provided the contention of the execution is in $O(\log\log n)$. This implies the next theorem:

▶ **Theorem 18.** *Any implementation of* graph-based-set *using any combination of* 1*-revealing primitives has an execution of $\dot{c}$ processes, each performing one* add *operation, such that the processes collectively take $\Omega(\dot{c}^2)$ steps, implying that the amortized (and hence, worst-case) step complexity of the* add *operation is $\Omega(\dot{c})$, provided $\dot{c} \in O(\log\log n)$.*

A data structure using a connected set of nodes is a graph-based-set, and therefore, Theorem 18 implies the lower bound also for these data structures.

▶ **Theorem 19.** *Any implementation of linked lists, skip lists, search trees and other data structures based on* graph-based-set*, using any constant set of* 1*-revealing primitives, has an execution of $\dot{c}$ processes, each performing one* add *operation, such that the processes collectively take $\Omega(\dot{c}^2)$ steps. Therefore, the amortized (and hence, worst-case) step complexity of the implementation is $\Omega(\dot{c})$, provided $\dot{c} \in O(\log\log n)$.*

Acknowledgements. We would like to thank the referees for many helpful comments.

References

1 James H Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2):75–110, 2003.

2 Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 370–380, New York, NY, USA, 1991. ACM. doi:10.1145/103418.103458.

3 Hagit Attiya and Arie Fouren. Poly-logarithmic adaptive algorithms require revealing primitives. *Journal of Parallel and Distributed Computing*, 109:102–116, 2017. doi:10.1016/j.jpdc.2017.05.010.

4 Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta informatica*, 9(1):1–21, 1977.

5 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, New York, NY, USA, 2014. ACM. doi:10.1145/2611462.2611486.

6 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM. doi:10.1145/1011767.1011776.

7 Timothy Harris. A pragmatic implementation of non-blocking linked-lists. *Distributed Computing*, pages 300–314, 2001.

8 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, pages 3–16. Springer-Verlag, 2005.

9 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

10 Yong-Jik Kim and James H Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

11 Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

12 Eric Ruppert. Analysing the average time complexity of lock-free data structures. In *Workshop on Complexity and Analysis of Distributed Algorithms*, 2016. [http://www.birs.ca/events/2016/5-day-workshops/16w5152/videos/watch/201612011630-Ruppert.html; accessed 22-Mar-2017].

13 Niloufar Shafiei. *Non-Blocking Data Structures Handling Multiple Changes Atomically*. PhD thesis, York University, 2015.

14 R. Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

15 P. Turan. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

16 John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

A Extending the Kim-Anderson Lower Bound

The $\Omega(\dot{c})$ lower bound proof on the worst-case step complexity for adaptive mutual exclusion [10], can be extended to yield an $\Omega(\dot{c})$ lower bound on the *amortized* step complexity.

Algorithm 1 Adaptive mutual exclusion using a set object and CAS

waitingRoom: a sack object, initially empty
promoted : a RW register, initially $\perp$
fastLane: CAS register, initially EMPTY
gate: CAS register, initially OPEN
privateGate$[0, \ldots n-1]$: initially CLOSED
wants$[0, \ldots, n-1]$: initially FALSE

```
1:  procedure enterCS(id)
2:      wants[id] = TRUE                          ▷ announce that the process needs to enter CS
3:      waitingRoom.put(id)                       ▷ add itself to the set of the waiting processes
4:      if CAS (fastLane, EMPTY, ⟨OCCUPIED, id⟩) then   ▷ try to occupy the fast lane
5:          continue   ▷ p_i successfully passed the fast lane, continue to the gate mutual exclusion
6:      else
7:          wait until privateGate[id] == OPEN    ▷ wait until another process
                                                  ▷ will open p_i's private gate
8:      end if
9:      wait until CAS (gate, OPEN, CLOSED)       ▷ try to win 2-process mutual exclusion
10: end procedure

11: procedure exitCS(id)
12:     wants[id] = FALSE
13:     privateGate[id] = CLOSED                                   ▷ clean-up
14:     CAS (fastLane, ⟨OCCUPIED, id⟩, EMPTY)   ▷ if the fast lane occupied by p_i, release it
15:     if promoted = id then                          ▷ if p_i was the promoted process
16:         promoted = ⊥                               ▷ the reset promoted to ⊥
17:     end if
18:     if promoted = ⊥ then          ▷ if there is no promoted process, promote one
19:         do next = waitingRoom.draw()               ▷ remove some process from the sack
20:         until (wants[next]==TRUE or next == ⊥)     ▷ until the removed process is interested
                                                       ▷ to enter CS, or until waitingRoom is empty
21:         if wants[next]==TRUE then                  ▷ process next is interested to enter CS
22:             promoted = next                        ▷ promote next
23:             privateGate[next] = OPEN   ▷ signal to next by opening its private gate
24:         end if
25:     end if
26:     gate = OPEN                                    ▷ release the 2-process mutual exclusion
27: end procedure
```

In each round of the proof of [10, Lemma 7], at most two processes are added to the set of *finished* processes $Fin(H)$. Before a process p is added to $Fin(H)$ in round j, it performs j steps. The execution H has k rounds, and there is at least one process that is not in $Fin(H)$ at the end of round k; this process performs at least k steps. Let x be the number of the processes in $Fin(H)$ at the end of the execution: $x = |Fin(H)|$. The processes $Fin(H)$ are simultaneously active in H, therefore $\dot{c}(H) \geq x$. The total number of steps performed by all processes in H is equal to the number of steps performed by the processes $Fin(H)$, which is at least $\sum_{j=1}^{x/2} 2j \geq x^2/4$, plus k steps performed by the last process. Thus, the total number of steps is $x^2/4+k$, and the amortized step complexity is $t(x) = (x^2/4+k)/(x+1) = \Omega(x) = \Omega(\dot{c}(H))$. Differentiating the last expression by x and equating to 0, we get that the amortized step complexity is minimized at $\min t(x) = \frac{1}{2}\left(\sqrt{4k+1}-1\right)$ for $x = \sqrt{4k+1}-1$. That is, the construction guarantees amortized step complexity $t(x) \geq x/2 = \Omega(\dot{c}(H))$.

The maximal number of processes in the system required for the construction is $N(k) = (2k+4)^{2(2^k-1)}$, implying $k = O(\log\log N)$. Since $x = O(\sqrt{k})$, we have that $x = O(\sqrt{\log\log N})$.

This gives the next observation:

▶ **Theorem 20.** *For any given x, there is an execution H, with point contention $\dot{c}(H) \geq x$ and RMR amortized step complexity $amortizedStep(H) = \Omega(\dot{c}(H))$, provided that $x = O(\sqrt{\log \log N})$, where N is the total number of the processes in the system.*

Mutual Exclusion Algorithms with Constant RMR Complexity and Wait-Free Exit Code

Rotem Dvir[1] **and Gadi Taubenfeld**[2]

1 **The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel**
`rotem.dvir@gmail.com`
2 **The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel**
`tgadi@idc.ac.il`

Abstract

Two local-spinning queue-based mutual exclusion algorithms are presented that have several desired properties: (1) their exit codes are wait-free, (2) they satisfy FIFO fairness, (3) they have constant RMR complexity in both the CC and the DSM models, (4) it is not assumed that the number of processes, n, is a priori known, that is, processes may appear or disappear intermittently, (5) they use only $O(n)$ shared memory locations, and (6) they make no assumptions on what and how memory is allocated.

The algorithms are inspired by J. M. Mellor-Crummey and M. L. Scott famous MCS queue-based algorithm [13] which, except for *not* having a wait-free exit code, satisfies similar properties. A drawback of the MCS algorithm is that executing the exit code (i.e., releasing a lock) requires spinning – a process executing its exit code may need to wait for the process that is behind it in the queue to take a step before it can proceed. The two new algorithms overcome this drawback while preserving the simplicity and elegance of the original algorithm.

Our algorithms use exactly the same atomic instruction set as the original MCS algorithm, namely: read, write, fetch-and-store and compare-and-swap. In our second algorithm it is possible to recycle memory locations so that if there are L mutual exclusion locks, and each process accesses at most one lock at a time, then the algorithm needs only $O(L+n)$ space, as compared to $O(Ln)$ needed by our first algorithm.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation

Keywords and phrases Mutual exclusion, locks, local-spinning, cache coherent, distributed shared memory, RMR complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.17

1 Introduction

Concurrent access to resources shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are still the de facto mechanism for concurrency control on shared resources: a process accesses the resource only inside a critical section code, within which the process is guaranteed exclusive access. The popularity of this approach is largely due to the apparently simple programming model of such locks, and the availability of lock implementations which are reasonably efficient.

Most of the mutual exclusion lock algorithms include busy-waiting loops. The idea is that in order to wait, a process *spins* on a flag register, until some other process terminates the spin with a single update operation. Unfortunately, under contention, such spinning may generate lots of traffic on the interconnection network between the process and the memory, which can slow other processes. To address this problem, it is important to distinguish

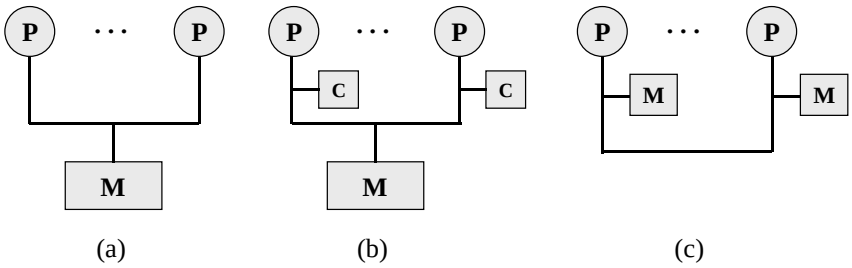


Figure 1 Shared memory models. (a) Central shared memory. (b) Cache Coherent (CC). (c) Distributed Shared Memory (DSM). P denotes processor, C denotes cache, M denotes shared memory.

between *remote* access and *local* access to shared memory, and to try to reduce the number of remote accesses as much as possible.

We consider two machine architectures models: (1) Cache coherent (CC) systems, where each process (or processor) has it own private cache. When a process accesses a shared memory location a copy of it migrates to a local cache line and becomes locally accessible until some other process updates this shared memory location and the local copy is invalidated; (2) Distributed shared memory (DSM) systems, where instead of having the "shared memory" in one central location, each process "owns" part of the shared memory and keeps it in its own local memory. These different shared memory models are illustrated in Figure 1.

A shared memory location is locally accessible to some process if it is in the part of the shared memory that physically resides on that process local memory. Spinning on a remote memory location while its value does not change, is counted only as *one* remote operation that causes communication in the CC model, while it is counted as *many* operations that causes communication in the DSM model. An algorithm satisfies *local spinning* (in the CC or DSM models) if the only type of spinning required is local spinning.

An algorithm that satisfies local spinning in a DSM system, is expected to perform well also when executed on a machine with *no* DSM. The reason is that each process spins only on memory locations on which no other process spins, thus eliminating hot-spot contention caused by busy-waiting.

The MCS lock, due to John Mellor-Crummey and Michael Scott, is perhaps the best-known and most influential local-spinning lock algorithm [13]. This important algorithm and several variants of it are implemented and used in various environments. For example, Java Virtual Machines use object synchronization based on variations of the MCS lock [7].

A code segment in an algorithm is *wait-free* if its execution by a process should require only a finite number of steps and must always terminate regardless of the behavior of the other processes. A drawback of the MCS lock is that releasing it is *not wait-free* and requires spinning – a process that is releasing the lock may need to wait for the process that is trying to acquire the lock to take a step before it can proceed. Thus, when there is high contention, a releasing process may have to wait for a long time until a process that is trying to acquire the lock is schedule. We present two new local-spinning algorithms which overcome this drawback while preserving the simplicity and elegance of the original MCS algorithm.

The two new mutual exclusion algorithms, which are inspired by the MCS algorithm, have several desired properties. These properties, formally defined in the next section, are: (1) their exit codes are wait-free, (2) they satisfy FIFO fairness, (3) they have constant RMR (remote memory reference) complexity in both the CC and the DSM models, (4) they do not require to assume that the number of participating processes, n, is a priori known, that is,

processes may appear or disappear intermittently, (5) they use only $O(n)$ shared memory locations, and (6) they make no assumptions on what and how memory is allocated[1].

Except for property 1 above, the other properties are satisfied also by the MCS algorithm. No previously published algorithm satisfies all these properties together.

Our algorithms use exactly the same atomic instruction set as the original MCS algorithm, namely: read, write, fetch-and-store and compare-and-swap. In our second algorithm it is possible to recycle memory locations so that if there are L locks, and each process accesses at most one lock at a time, then the algorithm needs only $O(L + n)$ space, as compared to $O(Ln)$ needed by our first algorithm.

2 Preliminaries

2.1 Computational model

Our model of computation consists of an asynchronous collection of n deterministic processes that communicate via shared registers (i.e, shared memory locations). Asynchrony means that there is no assumption on the relative speeds of the processes. Access to a register is done by applying operations to the register. Each operation is defined as a function that gets as arguments one or more values and registers names (shared and local), updates the value of the registers, and may return a value. Only one of the arguments may be a name of a *shared* register. The execution of the function is assumed to be atomic. Call by reference is used when passing registers as arguments. The operations used by all our algorithms are:

- *Read:* takes a shared registers r and simply returns its value.
- *Write:* takes a shared registers r and a value *val*. The value *val* is assigned to r.
- *Fetch-and-store* (FAS): takes a shared register r and a local register ℓ, and atomically assigns the value of ℓ to r and returns the previous value of r. (The fetch-and-store operation is also called *swap* in the literature.)
- *Compare-and-swap* (CAS): takes a shared register r, and two values: *new* and *old*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise r is left unchanged and the value false is returned.

Most modern processor architectures support the above operations.

2.2 Mutual exclusion

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among n competing processes [3]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical and exit. The entry section consists of two parts: the *doorway* which is *wait-free*, and the waiting part which includes one or more loops. A *waiting* process is a process that has finished its doorway code and reached the waiting part, and a *beginning* process is a process that is about to start executing its entry section. It is assumed that a process may crash[2] in its remainder section, but may not crash in its entry, critical or exit sections. It is also assumed that a process always leaves its critical section.

[1] For example, in [2] it is assumed that *all* allocated pointers must point to *even* addresses.
[2] A process that *fails by crashing* is a process that stops its execution in a definitive manner.

The *mutual exclusion problem* is to write the code for the entry and the exit sections in such a way that the following *two* basic requirements are satisfied.

- *Deadlock-freedom*: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- *Mutual exclusion*: No two processes are in their critical sections at the same time.

Satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm. For an algorithm to be fair, satisfaction of an additional condition is required.

- *FIFO (First-in-first-out)*: A beginning process cannot execute its critical section before a waiting process executes its critical section.
- *Strong FIFO*: A process that has not completed its doorway cannot execute its critical section before a waiting process executes its critical section.

All our algorithms satisfy the slightly stronger *strong FIFO* requirement. To simplify the presentation, when the code for a mutual exclusion algorithm is presented, only the entry code and exit code are described, and the remainder code and the infinite loop within which these codes reside are omitted.

2.3 Counting Remote Memory References

As already mentioned, for certain shared memory systems, it makes sense to distinguish between *remote* and *local* access to shared memory. Shared registers may be locally-accessible as a result of coherent caching, or when using distributed shared memory where shared memory is physically distributed among the processors.

We define a *remote reference* by process p as an attempt to reference (access) a memory location that does not physically reside on p's local memory. The remote memory location can either reside in a central shared memory or in some other process' memory.

Next, we define when remote reference causes *communication*. (1) In the *distributed shared memory* (DSM) model, any remote reference causes communication; (2) in the *coherent caching* (CC) model, a remote reference to register r causes communication if (the value of) r is not (the same as the value) in the cache. That is, communication is caused only by a remote write access that overwrites a different process' value or by the first remote read access by a process that detects a value written by a different process.

Finally, we define time complexity when counting only remote memory references. This complexity measure, called RMR complexity, is defined with respect to either the DSM model or the CC model, and whenever it is used, we will say explicitly which model is assumed.

- *The RMR complexity* in the CC model (resp. DSM model) is the maximum number of remote memory references which cause communication in the CC model (resp. DSM model) that a process, say p, may need to perform in its entry and exit sections in order to enter and exit its critical section since the last time p started executing the code of its entry section.

3 The First Algorithm

Our first algorithm has the following properties: (1) its exit code is wait-free, (2) it satisfies strong FIFO fairness, (3) it has constant RMR complexity in both the CC and the DSM models, (4) it does not require to assume that the number of participating processes, n, is a priori known, (5) it uses only $O(n)$ shared memory locations, (6) it makes no assumptions on what and how memory is allocated, and (7) it uses exactly the same atomic instruction set as the original MCS algorithm.

3.1 An informal description

The algorithm maintains a queue of processes which is implemented as a linked list. Each *node* in the linked list is an object with pointer field called *next*, boolean field called *locked*, and status bit called *status*. Each process p has its own *two* nodes (i.e., elements), called $q_p[0]$ and $q_p[1]$, which in a DSM machine can be assumed to be stored in process p's local memory. In addition, a shared object called T (tail), points to the end of the queue.

Each time a process p wants to enter its critical section it uses alternately one of its two nodes. In its entry code a process threads itself (i.e., its node) to the end of the queue. Afterwards, p checks its state which can be one of the following: (1) it is alone in the queue, (2) its predecessor is in its exit section, or (3) its predecessor is either in its entry or critical section. In the first two cases, p can safety enter its critical section, in the later case p spins locally on its boolean *locked* field until it gets a signal from its predecessor that it is now at the head of the queue. Once p is at the head of the queue it can enter its critical section.

In its exit code, a process signals to its successor to enter its critical section. The main challenge is in implementing the part of the algorithm in which a releasing process signals its successor, since the threading cannot be done in one atomic operation and requires several remote accesses to the shared memory. This includes making T point to this process' node and making the process' predecessor know the threaded process is its successor.

In the MCS algorithm, to prevent a race condition, the releasing process is required in its exit code to wait until the threading is completed, and only then it may signal its successor and exit. As a result, the exit code of the MCS algorithm is *not* wait-free. To resolve this problem, we had to deal with a situation where the releasing process is in its exit section, but since the threading of its successor has not been completed yet the releasing process does not know who is its successor and thus has no way to signal anything directly to its successor (unless it waits for the threading to be completed, which is not an option in our case).

So, in its exit code p first assigns the value *unlocked* to its *status* variable. Since p may not know who is its successor, this assignment leaves a signal for p's successor that it may enter its critical section. However, this signal is done in p's memory space, so its successor cannot simply spin and wait for this signal, and checks for this signal only once.

Then, p checks if the threading of its successor (if there is one) is completed. If it isn't, there are two possibilities: (1) p is alone in the queue in which case p completes its exit code, or (2) p is not alone, in which case its successor will check and notice the signal p left in p's *status* variable. If the threading of its successor is completed, again there are two options: (1) the successor has already noticed the signal in p's *status* variable in which case p completes its exit code, or (2) its successor is spinning locally on its *locked* bit, in which case p terminates the waiting by setting its successor's *locked* bit to false. There are several race conditions which are resolved using `compare-and-swap` operations as explained later. The reason for using two nodes for each process is explained in details in Subsection 3.3.

3.2 The code of the algorithm and a detailed description

In the algorithms, the following symbols are used: "$*$" to indicate pointer of a specified type, "&" to obtain an object's address, and "." (dot) for integrated pointer dereferencing and field access. The code of the algorithm appears in Figure 1. A detailed explanation follows.

We start with the entry code. In **line 1**, out of its two nodes, p chooses the node to use in the current iteration, by inspecting *current*'s value. We notice that when p finishes the iteration, it toggles the value of *current* (in **line 16**). In **line 2**, p initializes its current node *next* pointer to NIL. During the execution *next* points to p's successor in the queue, and p

Algorithm 1 Program for process p.

```
Type:   QNode: {next: QNode*, locked: bool, status ∈ {LOCKED, UNLOCKED}}
Shared: T: type QNode*, initially NIL          // T points to the last item in the queue
        q_p[0,1]: type QNode, both nodes initially {NIL, false, LOCKED}
            // queue nodes belong to process p, and local to process p in the DSM model
Local:  pred: type QNode*, initial value immaterial          // process' predecessor
        succ: type QNode*, initial value immaterial          // process' successor
        mynode: type QNode*, initially value immaterial      // currently used node
        current: ∈ {0,1}, initial value immaterial           // index to current node

    Enter Code:
1   mynode := &q_p[current]          // current node for this round, doorway begins
2   mynode.next := NIL
3   mynode.status := LOCKED
4   pred := FAS(T, mynode)           // enter the queue, doorway ends
5   if pred ≠ NIL then               // enter CS if no predecessor
6     mynode.locked := true          // prepare to wait
7     pred.next := mynode            // notify your predecessor
8     if CAS(pred.status, UNLOCKED, LOCKED) = false then
9       await (mynode.locked ≠ true) fi fi   // wait for your predecessor's signal
    Critical Section
    Exit Code:
10  mynode.status := UNLOCKED        // notify successor it can enter its CS
11  if mynode.next = NIL then        // if you don't have a successor
12    CAS(T, mynode, NIL)            // set T back to NIL if you are last
13  else if CAS(mynode.status, UNLOCKED, LOCKED) then   // there is a successor
14        succ := mynode.next
15        succ.locked := false fi fi   // notify successor it can enter its CS
16  current := 1 − current           // toggle for further use
```

has no successor yet. Later, the successor of p, if there is one, will update p's *next* pointer in **line 7**, and p will identify whether it has a successor, when executing **line 11**. In **line 3**, p initializes its current node *status* field to LOCKED. In **line 4**, p gets T's value, and assigns a pointer to its node into T. This line is the last line of the doorway, and it is where p threads its node to the queue and gets a pointer to its predecessor node (if there is one). In **line 5**, p validates whether it has a predecessor. If it doesn't, it means that p is first in the queue and can safely enter its critical section. If p has a predecessor, say q, it continues to **line 6**, where it initializes its *locked* variable to *true*, which means p cannot enter its critical section at the moment. In **line 7**, p notifies q that p itself is its successor. In **line 8**, p checks if q already enabled it to enter its critical section, by assigning UNLOCKED to *status*. If the *compare-and-swap* operation succeeds, p knows q already executed **line 10**, and exited its critical section, so p can enter its critical section. In case the *compare-and-swap* fails, p waits for its turn to enter its critical section in **line 9** by local-spinning on its *locked* variable, waiting for q to assign false to it (**line 15**).

Next we explain the exit code. In **line 10**, p assigns UNLOCKED to its *status* variable immediately after it finishes executing its critical section. At that time, p may not know who is its successor, so the first operation in the exit code is to leave a signal for the upcoming

successor, so that it will be able to enter its critical section when the time comes. In **line 11**, p checks if its successor (if there is one) has already notified who it is (that is, if its successor already executed **line 7**). If it didn't, p may be the only one in the queue. In **line 12** p checks whether T equals *mynode*, which is the node p inserted to the queue in **line 4**. If the *compare-and-swap* succeeds, p is indeed alone in the queue, so it assigns NIL to T, which returns the queue to its initial state. If the *compare-and-swap* in **line 12** fails, it means that there must be another process in the queue after p. Since p assigned UNLOCKED to its *status* variable, its successor should notice it and be able to enter its critical section later on. If p's successor has executed **line 7** before p has executed **line 11**, p will know who its successor is. In **line 13**, p checks whether its successor already let itself enter into its critical section after reading p's UNLOCKED value in **line 8**. If the successor hasn't done so yet, the *compare-and-swap* operation succeeds, and p lets its successor enter its critical section in **lines 14-15** by setting the bit its successor spins on to false. In **line 16**, p toggles its *current*, for the next iteration. The toggle is necessary to avoid deadlock.

3.3 Further explanations

In order to better understand the algorithm, we explain below three delicate design issues which are crucial for avoiding deadlocks.

1. *Why each process p needs two nodes $q_p[0]$ and $q_p[1]$?* The *current* variable is used to avoid deadlock in the following execution: assume each process has one node instead of two. Suppose process p is in its critical section, and process q finished its doorway. p resumes and executes its exit code (lines 10, 11). p finishes its exit code while q is in the queue, but q hasn't informed p who it is yet. p leaves its *status* variable with the value UNLOCKED, so that q will be able to enter its critical section. p starts another iteration before q resumes and executes line 4. Another process q' executes its entry code, such that q' is p's successor. Notice that, in that execution, q and q' share p's node as their predecessor (from two different iterations of p). If q' executes line 7 before q, q can override the assignment of q' and assigns a pointer to its node into p's *next* variable. q' now moves on and waits in line 9, but there is no process to free q' and a deadlock occurs. This problem is resolved by having each process own two nodes. We only need two nodes for each process, since the algorithm satisfies FIFO. p's successor enters its critical section before p enters its critical section in its next iteration. After p's successor enters its critical section, p's node won't be needed anymore, and p will be able to reuse it.
2. *Is the order of lines 6 and 7 important?* Line 6 must be executed before line 7, and their order should not be changed. Assume we have two processes, p and q, where q is p's predecessor and we change the order of lines 6 and 7. We let p execute line 7 and suspend it before executing line 6. In such a case, q can start executing its exit code. q will notice it has a successor, and q will move on to execute line 15. Then, p will continue to execute its code, and executes line 6. p assigns *true* to *locked* and misses q's signal to enter its critical section, and a deadlock occurs.
3. *Is the order of lines 10 and the test in the if statement in line 11 important?* Line 10 must be executed before the exited process checks in line 11 whether it has a successor. If the UNLOCKED assignment is executed afterwards, a deadlock may occur. Assume we have two processes, p and q, where q is p's predecessor. q executes the first line of the exit code, which is (after switching) "**if** *mynode.next* = NIL **then**". Assume p hasn't executed line 7 yet, so the condition is true. Meanwhile, process p executes lines 5-9 and waits at line 9. Process p cannot skip the *compare-and-swap* since q's *status* variable is LOCKED. q finishes its exit code without signaling to p, and thus p will spin in line 9 forever, causing a deadlock.

3.4 Correctness proof

The following notions and notations are used in the proof.

1. **Doorway**: Process p is considered to be in its *doorway* while executing statements 1-4.
2. **The i^{th} iteration**: Process p during its i^{th} iteration (i.e, its i^{th} attempt to enter its critical section) is denoted by p^i.
3. **Follows, predecessor, successor**: Consider an execution e. q^j *follows* p^i in e if and only if p^i finishes its doorway before q^j. p^i is the *predecessor* of q^j in e if and only if q^j follows p^i, and no other process finishes its doorway between the time p^i finished its doorway to the time q^j finishes its doorway. If p^i is the *predecessor* of q^j then q^j is said to be the *successor* of p^i.

▶ **Lemma 1.** *For every process p at iteration i, p^i has at most one predecessor.*

Proof. The fact that a process may have only a single predecessor, follows from that fact that the last step of the doorway (line 4) is an *atomic* fetch-and-store operation which updates T and *pred*. ◀

▶ **Lemma 2.** *Assume that for every p^i and q^j, if p^i is the predecessor of q^j then p^i enters its critical section before q^j enters its critical section. Then, for every p^i and q^j, if p^i is the predecessor of q^j then p^i enters its critical section before any process that follows q^j enters its critical section.*

Proof. Proof by induction on the number of processes m that follow q^j. In the base case, when $m = 1$, there is only one process, say r^k, that follows q^j. This means that q^j is the predecessor of r^k. Therefore, according to the assumption made in the first part of the lemma, q^j enters its critical section before r^k enters its critical section. Since p^i enters its critical section before q^j, and q^j enters its critical section before r^k, by transitivity p^i enters its critical section before r^k.

We assume that the lemma holds for $m-1$ processes that follow q^j, and prove that it also holds for the m^{th} process that follows q^j. Let the m^{th} process be r^k. We denote r^k's predecessor as $\hat{r^k}$. Notice that $\hat{r^k}$ is the $m-1$ process that follows q^j. Thus, by the induction hypothesis, p^i enters its critical section before $\hat{r^k}$ enters its critical section. $\hat{r^k}$ is the predecessor of r^k, and thus, according to the assumption made in the first part of the lemma, $\hat{r^k}$ enters its critical section before r^k enters its critical section. Since p^i enters its critical section before $\hat{r^k}$ enters its critical section and $\hat{r^k}$ enters its critical section before r^k enters its critical section, by transitivity p^i enters its critical section before r^k. ◀

▶ **Lemma 3.** *For every p^i and q^j such that p^i is the predecessor of q^j, p^i is the only process that can assign false to q^j's mynode.locked.*

Proof. By Lemma 1, q^j has at most one predecessor. So, p^i is q^j's only predecessor. Clearly, except for p^i, any other process that q^j follows will not be able to write to q^j's *mynode.locked*. Thus, throughout the algorithm, the only processes that write to q^j's *mynode.locked* are q^j itself and p^i. q^j does it at line 6 and p^i does it at line 15. At line 6, q^j assigns true to *locked*, thus, the only process that assign false is p^i. ◀

▶ **Lemma 4** (STRONG FIFO). *If p^i finishes its doorway before q^j (begins or) finishes its doorway, then p^i enters its critical section before q^j enters its critical section.*

Proof. p^i finished the doorway before q^j finishes the doorway, therefore p^i executes line 4 before q^j does. There are two options:

1. q^j is p^i's successor.
2. q^j follows p^i, but q^j is not p^i's successor.

According to Lemma 2, once we prove that p^i enters its critical section before q^j in case 1, then it would immediately follow that p^i enters its critical section before q^j in case 2 as well.

Assume q^j is p^i's successor and assume to the contrary that q^j enters its critical section before p^i. There are two cases:

1. q^j executes line 4 after p^i executes line 12. In which case, p^i entered its critical section before q^j, which contradicts the assumption.
2. q^j executes line 4 before p^i executes line 12. Therefore, T still points to p^i's *qnode* when q^j executes line 4 and q^j gets p^i's *qnode* to its *pred* variable. Thus *pred* is not NIL, and q^j continues and executes line 8. Here we have two options as well:
   a. The *compare-and-swap* operation in line 8 succeeds. The *compare-and-swap* succeeds only if q^j's *pred.status* is equal to UNLOCKED. *pred* is p^i's *qnode*, p^i assigned its status LOCKED value at the beginning of p^i's entry code. For it to change to UNLOCKED, p^i should execute line 10 in the exit code. This means that p^i entered its critical section before q^j did, contradicting the assumption.
   b. The *compare-and-swap* operation in line 8 fails. q^j continues to line 9 and local-spins on *locked*. The only way q^j can enter its critical section is when some other process writes false to *locked*. According to Lemma 3, p^i is the only process that can assign false to q^j's *locked*. Notice that the only line where p^i does this is at line 15, which is in its exit code. In this case, p^i entered its critical section before q^j, contradicting the assumption.

We proved that if p^i finishes the doorway before q^j finishes the doorway, then there is no valid scenario where q^j enters its critical section before p^i. Therefore, p^i enters its critical section before q^j does, implying that the algorithm satisfies strong FIFO. ◀

▶ **Lemma 5.** *For every p^i and q^j, if p^i is the predecessor of q^j such that p^i and q^j are not in their critical sections at the same time, then p^i and r^k are not in their critical sections at the same time, for any other process r^k that follows q^j.*

Proof. Proof by induction on the number of processes m that follow q^j. In the base case, $m = 1$: there is only one process, say r^k, that follows q^j. This means that q^j is the predecessor of r^k. Assume p^i is in its critical section. Since q^j is not in its critical section at the same time as p^i and by Lemma 4 the algorithm satisfies strong FIFO, therefore q^j enters its critical section after p^i. Also by Lemma 4, r^k enters its critical section after q^j. Therefore, r^k cannot be in its critical section at the same time as p^i. Next, we assume that the lemma holds for $m-1$ processes that follow q^j, and prove for m processes that follow q^j. Let the m^{th} process be r^k, and let r^k's predecessor be $r^{\hat{k}}$. Notice that $r^{\hat{k}}$ is the $m-1$ process following q^j, and therefore by the induction hypothesis p^i and $r^{\hat{k}}$ are not in their critical sections at the same time. Thus, since by Lemma 4 the algorithm satisfies strong FIFO, r^k is not in its critical section at the same time as p^i. ◀

▶ **Lemma 6** (Mutual Exclusion)**.** *No two processes are in their critical sections at the same time.*

Proof. We assume that there are two processes, p^i and q^j at their critical sections at the same time, and show it leads to a contradiction. Assume, without loss of generality, that p^i enters its critical section before q^j. According to Lemma 4 the algorithm satisfies strong FIFO, so p^i finishes its doorway before q^j, and therefore p^i executes line 4 before q^j. There are two options:

1. q^j is p^i's successor.
2. q^j follows p^i, but q^j is not p^i's successor.

By Lemma 5, once we prove that p^i and q^j are not in their critical sections at the same time in case 1, then it would immediately follow that p^i and q^j are not in their critical section at the same time in case 2. Let's prove case 1: Assuming q^j is p^i's successor, there are two options:

1. q^j executes line 4 after p^i executes line 12. In this case, p^i exits its critical section before q^j enters its critical section, which contradicts the assumption.
2. q^j executes line 4 before p^i executes line 12. Therefore, T still points to p^i's *qnode* when q^j executes line 4 and q^j gets p^i's *qnode* assigned to its *pred* variable. Thus *pred* is not NIL, and q^j continues and executes line 8. In line 8, q^j checks whether p^i's status is UNLOCKED. Since p^i executed line 4 before q^j executed line 4, p^i also executed line 3 before q^j executed line 8. According to the assumption p^i still hasn't exited its critical section, so its status is still LOCKED. Therefore the *compare-and-swap* fails and q^j continues to line 9. q^j assigned true to its *locked* variable in line 6, and local-spins in line 9, waiting for some process to let it enter its critical section. By Lemma 3, p^i is the only process that can assign false to q^j's *locked*. p^i does this at line 15, which means that for q^j to enter its critical section, p^i has to execute its exit code. This contradicts the assumption that p^i and q^j are in their critical sections at the same time. ◀

▶ **Lemma 7.** *For every p^i and q^j such that q^j is p^i's predecessor, once q^j assigned the value false to p^i's locked variable, this value cannot be overwritten, until p^i completes its exit code.*

Proof. By Lemma 1, p^i has only one predecessor, so q^j is p^i's only predecessor. Throughout the algorithm, the only processes that write to p^i's *locked* variable are p^i itself and q^j. p^i does this at line 6 and q^j does this at line 15. We will show that if q^j executes line 15, p^i must have already executed line 6 before that. q^j executed line 15, therefore q^j must have executed the *"else"* clause of the *"if"* statement at line 11. So the *"if"* condition was false, and q^j's *mynode.next* was not equal to NIL. q^j has only one successor which is p^i. Since q^j's *next* variable was not NIL, p^i has already executed line 7 and assigned its node to q^j's *next*. This implies that p^i has already executed line 6, in which it assigns true to *locked*. Therefore p^i executed line 6 before q^j executed line 15, and since there is no other process that writes to p^i's *locked* variable, the (false) value was not overwritten. ◀

▶ **Lemma 8** (Deadlock-freedom)**.** *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

Proof. Assume to the contrary that some group of processes P are in their entry code and none of them can ever access its critical section. Let p^i be the first process in P to complete its doorway. Thus, all the other processes in P follow p^i. The fact that p^i is not being able to enter its critical section, means that p^i is local-spinning at line 9, since all the other lines in its entry code do not contain any loops, are wait-free and can be completed in a constant number of p^i steps. Any other execution path would lead p^i to its critical section and contradicts the assumption. From here it follows that:

- When p^i completed its doorway, *pred* was not NIL. Therefore, there exists process q^j such that q^j finished the doorway before p^i, but hasn't executed line 12 that removes it from the queue.
- p^i *compare-and-swap* operation at line 8 fails, therefore q^j *status* was equal to LOCKED. There are two options:
  1. q^j hasn't executed line 10.
  2. q^j has already executed line 13, where the *compare-and-swap* operation ended successfully for q^j.

Since we assumed that p^i is the first process to complete its doorway among the waiting processes, and q^j is p^i's predecessor, it follows that q^j would eventually be able to enter its critical section. Therefore, q^j will necessarily exit its critical section and begin the exit code. In both cases above, it is easy to see that the execution path of q^j in the exit code leads it to line 15, where q^j writes false to p^i's *locked* variable. In addition, according to Lemma 7, the value of p^i's *mynode.locked* field is never overwritten until p^i completes it exit code. Therefore, at some point p^i will notice that $mynode.locked = false$ and can continue to its critical section. This contradicts the assumption that p^i will not enter its critical section. ◀

▶ **Lemma 9** (Constant RMR complexity). *The RMR complexity of Algorithm 1 is $O(1)$ in both the CC and DSM models.*

Proof. By inspecting the algorithm, it is easy to count steps and see that except the busy-waiting loop in line 9, it takes constant number of steps for a process to enter and exit its critical section. Thus, it is sufficient to prove that, for every p^i, p^i performs $O(1)$ RMRs at line 9, because this is the only busy-waiting loop in the algorithm. We will prove that while the process is executing the loop at line 9, it performs only a constant number of remote memory references, in both models:

- **DSM model**: p^i spins on *mynode.locked*. *mynode* can be equal to either $q_p[0]$ or $q_p[1]$ as follows from line 1 in the algorithm. Both of them initialized as local to process p^i's memory and thus the algorithm performs $O(1)$ RMRs in the DSM model.
- **CC model**: We prove that in one iteration of a process, there is at most one cache invalidation. Before spinning on *mynode.locked*, its value migrates to p^i's local cache, since p^i assigned to it at line 6. It is updated by another process only once, at line 15. When a process updates *mynode.locked*, p^i will have a cache invalidation and p^i will execute one remote memory reference to read the new value of *mynode.locked*. Since the new value is necessarily equal to false, p^i stops spinning on *mynode.locked* and proceeds to its critical section. By Lemma 7, there is no other process that writes to p^i's *mynode.locked* in the current iteration anywhere else in the algorithm. Therefore, there is only one remote memory reference during the loop execution, and the algorithm has $O(1)$ RMR complexity in the CC model. ◀

▶ **Lemma 10** (Wait-free exit code). *Every process finishes its exit code within a bounded number of its own steps.*

Proof. Since the exit code is a straight-line code which does not contain either loops or await operations, it immediately follows that any execution of the exit code will be completed in a bounded number of a process' own steps. ◀

▶ **Theorem 11.** *Algorithm 1 satisfies mutual exclusion, deadlock freedom, wait-free exit, strong FIFO fairness, and constant RMR complexity. Furthermore, it does not require to assume that the number of processes, n, is a priori known, it uses only $O(n)$ shared memory locations, it makes no assumptions on what and how memory is allocated, and it uses exactly the same atomic instruction set as the original MCS algorithm.*

Proof. The properties mutual exclusion, deadlock freedom, wait-free exit, strong FIFO fairness and constant RMR complexity, follows from Lemma 6, Lemma 8, Lemma 10, Lemma 4 and Lemma 9, respectively. The other properties are easily verified by inspecting the code of the algorithm. ◀

4 The Second Algorithm

The second algorithm satisfies the same properties as the first algorithm, as listed at the beginning of Section 3. In addition, in the second algorithm it is possible to recycle memory locations so that if there are L locks, and each process accesses at most one lock at a time, the algorithm needs only $O(L+n)$ space, as compared to $O(Ln)$ needed by the first algorithm.

To simplify the presentation, we will assume that processes have unique identifiers. However, for each process p, the value of one of its pointers is a unique number which can be used as process p's unique identifier instead of assuming that p itself is the unique identifier. We elaborate more on this issue in Subsection 4.3, after the algorithm is presented.

4.1 An informal description

As in the previous case, the algorithm maintains a queue of processes which is implemented as a linked list. Each process p has two *different* data elements that complete each other and together represent a single node. The two data elements are called: q_p which resides in p local memory and access to it is considered local access, while $node_p$ is handed over from an exiting process to its successor at the end of the exit code, and access to it is considered remote memory access. q_p, which is not part of the queue, is a record with pointer field called *qnode* which initially points to the $node_p$ element, and boolean field called *locked*. $node_p$ is the element that initially p tries to thread into the linked list in its entry code.

In addition, a shared object called T (tail), points to the end of the queue. Initially, when the queue is empty, T points to a *dummy* node, called $node_0$, that enables the first process which succeeds to enter the queue, to proceed to its critical section.

In its entry code a process threads itself (i.e., thread the element $q_p.qnode$ points to) to the end of the linked list. A process has several ways to enter its critical section: it can enter immediately if it is alone in the queue or if its predecessor is in its exit section, otherwise, it has to spin locally until its predecessor assigns false to *locked*.

In the exit code p first assigns the value p (its ID which is different than 0) to its *status* field of its node in the linked list. Since, p may not know who is its successor, this assignment leaves a signal for p's potential successor that it may enter its critical section. Then, p checks if it has a successor. If it doesn't then p completes its exit code. Otherwise, if p's successor has already noticed p's *status* variable equals p, process p completes its exit code. If its successor is spinning locally on its *locked* bit, p terminates the waiting by setting its successor *locked* bit to false. In *all* the above cases, p always leaves its current node in the queue (because this node includes the *status* field with the value p which indicates that its critical section is free) and, before p completes its exit code, it removes from the queue and takes ownership of the node of its predecessor (which is the current dummy node).

4.2 The code of the algorithm and a detailed description

We assume that each process has a unique identifier which is different than 0. The code of the algorithm appears in Figure 2. It is important to notice that there are some statements in the algorithm with multiple memory references. We use this style to keep the algorithm short and simple. Only one shared memory location can be accessed in one atomic step! For example, the statement in line 1, "$q_p.qnode.next$:= NIL" is equivalent to "$localTemp := q_p.qnode$; $localTemp.next$:= NIL", and the statement in line 7 "$pred.next := q_p.qnode$" is equivalent to "$localTemp := q_p.qnode$; $pred.next := localTemp$". A detailed explanation follows.

Algorithm 2 Program for process p.

Type: $QNode$: {next: QNode*, local: LocalNode*, status: integer, pid: integer}
$LocalNode$: {qnode: QNode*, locked: bool}
Constant:
ZERO = 0 // it is assumed that 0 is not a process id
Shared: $node_0$: type QNode, initially {NIL, NIL, 0, 0}
// a dummy node, enables the first process to enter its critical section
T: type QNode*, initially $\&node_0$ // T points to $node_0$
$node_p$: type QNode, initial values are immaterial
q_p: type LocalNode*, qnode initially $\&node_p$, locked initial value is immaterial
// q_p belongs to process p, and local to process p in the DSM model
Local: $pred$: type QNode*, initial value is immaterial // process' predecessor
$predPid$: type integer, initial value is immaterial // process' predecessor ID

Enter Code:
1 $q_p.qnode.next$:= NIL // initialization, doorway begins
2 $q_p.qnode.pid := p$ // process ids are unique and different than 0
3 $q_p.qnode.local = q_p$
4 $q_p.qnode.status$:= ZERO
5 $q_p.locked := true$
6 $pred$:= **FAS**$(T, q_p.qnode)$ // enter the queue, doorway ends
7 $pred.next := q_p.qnode$ // notify your predecessor
8 $predPid := pred.pid$
9 **if CAS**$(pred.status, predPid,$ ZERO$) = false$ **then**
10 **await** $(q_p.locked \neq true)$ **fi** // wait for your predecessor's signal
Critical Section
Exit Code:
11 $q_p.qnode.status := p$ // notify successor it can enter its CS
12 **if** $q_p.qnode.next \neq$ NIL **then** // if you have a successor
13 **if CAS**$(q_p.qnode.status, p,$ ZERO$)$ **then**
14 $q_p.qnode.next.local.locked := false$ **fi fi** // notify successor it can enter its CS
15 $q_p.qnode := pred$ // use predecessor's node for the next iteration

We start with the entry code. In **lines 1-5**, p initializes its node. Notice p initializes all of its fields except $q_p.qnode$, which was already initialized before the execution began. In **line 6**, p executes *fetch-and-store* to enter the queue. p gets T's value, which points to the last item in the queue (which is also p's predecessor), and assigns its $qnode$ to T. Notice that p assigns only its $node_p$ to T, while q_p can be accessed via $node_p$ if needed. This is the end of the doorway. In **line 7**, p notifies its predecessor that p is its successor. In **line 8**, p copies its predecessor's process ID to a local variable, to be used as an argument to the *compare-and-swap* operation later. In **line 9**, p checks whether its predecessor has already signaled p to enter its critical section. The predecessor does it by assigning its process ID to its *status* in **line 11**. If the predecessor already assigned its process ID but did not change it back to ZERO yet (**line 13**) then the *compare-and-swap* succeeds and p can enter its critical section. If it fails, p continues to **line 10** and starts spinning locally on its *locked* variable, waiting for it to change to false so it can enter its critical section.

Next we explain the exit code. In **line 11**, p assigns its process ID to its *status* variable, and signals its potential successor that it can enter its critical section. In **line 12**, p checks whether it has a successor. If *next* equals NIL it doesn't have a successor and it continues to **line 15**. If *next* is not NIL, it means that some process q already assigned itself as p's successor at **line 7**. In such a case, p continues to **line 13**, checking whether q has already executed **line 9** and let itself enter its critical section. If q executed **line 9** and q's *compare-and-swap* ended successfully, p's *compare-and-swap* in **line 13** fails and p continues to **line 15**. If p's *compare-and-swap* ends successfully, it means q hasn't entered its critical section yet. Since p assigned ZERO to its *status* in **line 13**, p must let q enter its critical section by setting *locked* to false, and does so at **line 14**. In **line 15**, p assigns its predecessor's node to itself, and leaves its node to its successor. On p's next iteration, it will use its predecessor's node, thus p will not override the *status* in its previous node when initializing its *qnode* at the beginning of its next iteration. Therefore, p's successor will be able to read and use p's *status* when needed and find out that it can enter its critical section.

4.3 Further explanations

In order to better understand the algorithm, we explain below several crucial design issues.

1. *Do we really need to explicitly assume that the processes have unique identifiers?* No, this is done only to simplify the presentation. In the first algorithm, it is not assumed that processes have unique identifiers. However, each process p has two unique memory nodes $q_p[0]$ and $q_p[1]$, and it is possible to consider $\&q_p[0]$ as the unique identifier of process p. Similarly, in algorithm 2, the value of the pointer q_p is a unique number which can be used as process p's unique identifiers. That is, in Algorithm 2, it is possible to replace p with q_p (assuming $q_p \neq 0$) everywhere (i.e., in lines 2,11,13). This implies that also in Algorithm 2 there is no need to explicitly assume that processes have unique IDs.
2. *How does a process that does not need to spin know that it is at the head of the queue?* Whenever the values of the *status* field and of the *pid* field, of the first node (i.e., the dummy node) in the queue are equal, the process its node is the successor of the dummy node can safely enter its critical section. Initially, $node_0$ is the dummy node and $node_0.status = node_0.pid = 0$, thus, the first process that threads itself into the queue, can immediately enter its critical section. Also, when a process, say p, completes its critical section, it always leaves its *current* node in the queue and takes ownership of the node of its predecessor (which is the dummy node). Thus, its current node becomes the new dummy node with both *status* and *pid* fields equal p, which will not block the next process in line, since the *compare-and-swap* operation in line 9 would succeed.
3. *Can't we simply use UNLOCKED in lines 2,11 and 13, as done in algorithm 1, instead of using the unique process identifier p?* No, this is crucial for avoiding deadlocks. We explain it by example. Consider the following scenario: There are two processes, p and q.
   - p starts first and enters its critical section;
   - q starts, run until after line 7 and becomes p's successor in the queue;
   - p executes the code until after line 12. Since $next \neq$ NIL, p enters the *if* statement;
   - q continues and enters its critical section;
   - q continues, finishes its exit code and takes p's current *qnode* for its next iteration;
   - q starts its entry code, and executes until after line 11; p and q has the same *qnode*!

   Now, here is the difference between using process identifiers and UNLOCKED: p continues,
   - When process identifiers are used, the *compare-and-swap* operation in line 13 *fails* and p takes its predecessor's *qnode* for its next iteration and completes its exit code.

- When UNLOCKED is used, the *compare-and-swap* operation in line 13 *succeeds* for p and the shared qnode for p and q now contains the status ZERO. p will continue to line 14, and p will complete its exit code. q executes its exit code and since it has no successor, it skips lines 12-14 and exits. We got into a situation where the queue is empty, and T points to a dummy node with status value ZERO! The next process to enter will be spinning in line 10 forever.

The structure of the correctness proof of Algorithm 2 is similar to that of Algorithm 1.

5 Related Work

Mutual exclusion algorithms were first introduced by Edsger W. Dijkstra in [3]. Since then, numerous implementations have been proposed [15, 7, 17]. The first queue-based local-spinning mutual exclusion algorithms for the CC model were presented in [1, 6]. The algorithm from [1] used the `fetch-and-increment` operation, while the algorithm from [6] used the `fetch-and-store` (swap) operation. In these two algorithms different processes may spin on the same memory location at the different times. Their RMR time complexity in the CC model is a constant, while their time complexity in the DSM model is unbounded.

The famous MCS algorithm is from [13]. Unlike the previous two algorithms, the MCS algorithm satisfies local spinning in *both* the CC model and the DSM model. In [9], a simple correctness proof of the MCS lock is provided. An extension of the MCS Algorithm that solves the readers-writers problem is presented in [14]. In [16] queue-based algorithm is presented, which uses unbounded space, in which it is possible for a spinning process to "become impatient" and leave the queue before acquiring the lock. A recoverable version of the MCS algorithm, in which processes can fail and recover, was presented recently [4].

Another queue-based lock was developed by Craig [2] and, independently by Magnusson, Ladin and Hagersten [11, 12]. As the MCS lock, the queue is implemented as a linked list, but with pointers from each process to its *predecessor*. The algorithm uses `fetch-and-set` operations and may outperform the MCS lock on cache-coherent machines. Its time complexity in the CC model is a constant, while its time complexity in the DSM model is unbounded.

A variant of the above algorithm from [2], with constant time complexity in both the CC and the DSM models was presented in [2]. For this variant to work, it must be assumed that *all* allocated pointers point to *even* addresses. This assumption enables to pack two shared registers into a single 32-bit word so that it is possible to atomically swap the two registers as a unit. In [10], four local-spin mutual exclusion algorithms for the DSM model using `fetch-and-set` operations were presented; all these algorithms use arrays of fixed size, assume that the number of processes is a priori known, and are not suitable for a model where processes may appear or disappear intermittently.

6 Discussion

We have presented two new mutual exclusion algorithms, that overcome a drawback of the famous MCS algorithm, while preserving its simplicity, elegance and properties. It would be interesting to design additional new algorithms, which would be based on our algorithms, that implement other types of locks, such as readers-writers locks [14], abortable locks [8, 16] and recoverable locks [4, 5], and would have constant RMR complexity, satisfy the wait-free exit code property and other desired properties.

References

1 T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

2 T.S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR-93-02-02, Dept. of Computer Science, Univ. of Washington, February 1993.

3 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

4 W. Golab and D. Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017.

5 W. Golab and A. Ramaraju. Recoverable mutual exclusion. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.

6 G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computers*, 28(6):69–69, June 1990.

7 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, 2008. 508 pages.

8 P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, pages 295–304, July 2003.

9 T. Johnson and K. Harathi. A simple correctness proof of the MCS contention-free lock. *Information Processing Letters*, 48(5):215–220, 1993.

10 H. Lee. Local-spin mutual exclusion algorithms on the DSM model using fetch&store objects. Mater thesis, University of Toronto, 2003.

11 P.S. Magnusson, A. Landin, and E. Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, February 1994.

12 P.S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the 8th International Symposium on Parallel Processing*, pages 165–171, April 1994.

13 J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.

14 J.M. Mellor-Crummey and M.L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices*, 26(7):106–113, 1991.

15 M. Raynal. *Algorithms for mutual exclusion.* The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.

16 M.L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proc. 21th ACM Symp. on Principles of Distributed Computing*, pages 31–40, July 2002.

17 G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

Remote Memory References at Block Granularity

Hagit Attiya[1] and Gili Yavneh[2]

1 **Department of Computer Science, Technion, Haifa, Israel**
`hagit@cs.technion.ac.il`
2 **Department of Computer Science, Technion, Haifa, Israel**
`giliyav@cs.technion.ac.il`

Abstract

The cost of accessing shared objects that are stored in remote memory, while neglecting accesses to shared objects that are cached in the local memory, can be evaluated by the number of *remote memory references* (*RMRs*) in an execution. Two flavours of this measure – *cache-coherent* (CC) and *distributed shared memory* (DSM) – model two popular shared-memory architectures. The number of RMRs, however, does not take into account the *granularity* of memory accesses, namely, the fact that accesses to the shared memory are performed in *blocks*.

This paper proposes a new measure, called *block RMRs*, counting the number of remote memory references while taking into account the fact that shared objects can be grouped into blocks. On the one hand, this measure reflects the fact that the RMR incurred for bringing a shared object to the local memory might save another RMR for bringing another object placed at the same block. On the other hand, this measure accounts for *false sharing*: the fact that an RMR may be incurred when accessing an object due to a concurrent access to another object in the same block.

This paper proves that in both the CC and the DSM models, finding an optimal placement is NP-hard when objects have different sizes, even for two processes. In the CC model, finding an optimal *placement*, i.e., grouping of objects into blocks, is NP-hard when a block can store three objects or more; the result holds even if the sequence of accesses is known in advance. In the DSM model, the answer depends on whether there is an efficient mechanism to inform processes that the data in their local memory is no longer valid, i.e., cache coherence is supported. If coherence is supported with cheap invalidation, then finding an optimal solution is NP-hard. If coherence is not supported, an optimal placement can be achieved by placing each object in the memory of the process that accesses it most often, if the sequence of accesses is known in advance.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases false sharing, cache coherence, distributed shared memory, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.18

1 Introduction

In a typical multiprocessor, processes communicate by concurrently accessing objects in the shared memory. To reduce the high cost of access to the shared memory, a fast memory, local to each process, is used to cache recently-used objects. The cost of a *cache hit* – finding an object in the process's local memory – is negligible relative to the cost of a *cache miss*, which requires an access to the shared memory. We assume that a block is not evicted from the local memory unless it is required by some other process; that is, the local memory is large and fully associative.

A *remote memory reference* (*RMR*) [25] is incurred for every cache miss, according to one of two models: In the *cache coherent* (*CC*) model, the first time a process accesses an

object it puts the object in its local memory; subsequent reads and writes to the same object by the same process are free, as long as no other process modifies the object between them; otherwise, the object must be updated in the local memory again, incurring an RMR. In the *distributed shared memory* (*DSM*) model, each object is created in the local memory of some process, before the first access to it occurs; an access to the local memories of other processes incurs one RMR, while an access to the local memory is free. If cache coherence and cheap data invalidation is supported then, like in the CC model, the object will be copied by other processes for read operations, while write operations invalidate the copies at other processes. Making a copy incurs an RMR, while accesses to an existing copy do not. If invalidation is not supported, the object remains in the local memory of the process that created it, and any access to it by a different process incurs an RMR.

Both models, however, ignore the fact that access to the shared memory is performed in *blocks*, namely, several objects are placed together and moved together between local memories and shared memory. When an object is read into the memory, all the objects in the same block are moved, so that later accesses to them incur a cache hit. Similarly, if one object is moved to another process, accesses to the other objects in the same block causes a cache miss, even if no other process has accessed them in between. This phenomenon is called *false sharing* [23, 11].

Many works deal with cache-conscious organization of the memory, namely, the placement of objects in blocks so as to increase the number of cache hits, e.g., [8, 9, 20]. However, as we discuss in Section 1, these works consider only single-process scenarios and do not take into account the effects of concurrent access to blocks. Algorithms and lower bounds on the number of RMRs, which capture the effects of concurrency, do not take into account the granularity of memory accesses, which are done in blocks.

Our Contributions

We introduce the *block RMRs* complexity measure. In the CC model, when a process accesses a block, an RMR is incurred when the block is brought to its cache, and later accesses to the same block are free, as long as no other process writes an object in this block. For this model, we prove that finding an optimal placement of objects into blocks, i.e., a placement with a minimal number of block RMRs, is *NP-hard* when blocks can hold three or more objects; the result holds even when the sequence of accesses is known and all objects have the same size (Section 3.2). The problematic access sequence is a natural one, in which two processes perform a *traversal* on a graph. When blocks can hold two objects of the same size and the access sequence is known, we present an efficient algorithm for placing objects in blocks for the block-based CC model (Section 3.1).

In the *block-based DSM* model, each block is created in a specific process when the execution starts, and each access to a block not in a process's own cache incurs an RMR. If cache coherence is supported then an object may have several copies, in which case, consecutive accesses to the same block are free, as long as no other process writes an object in this block. If cache coherence is not maintained, then the object remains in the local memory of the process that created it, and accesses from other processes incur an RMR, while accesses of the creating process are free.

For the block-based DSM model, we prove that the number of block RMRs depends on the cost of invalidation and existence of cache coherence. If invalidation cost is negligible and cache coherence is supported then finding a placement of objects into blocks is *NP-hard*, even if the access sequence is known and all objects have the same size. If cache coherence is not supported, we show that the number of block RMRs is indifferent to the order of accesses

in the sequence, and use this result to design an algorithm that finds a placement with an optimal number of block RMRs, when the access sequence is known in advance. (The results for the DSM model appear in Section 4.)

For both models, we prove that handling objects with varying sizes makes the problem NP-hard (Appendix A). This result is achieved with an access sequence in which two processes traverse a tree one after the other. Because of this result, the rest of the paper concentrates on the case where all objects have the same size.

Related work

Remote memory references have been proposed as a way to predict the scalability of shared-memory programs [25]. They have been applied for evaluating the complexity of mutual exclusion algorithms and presenting lower bounds for it, in the CC and DSM models (see the survey [3]). Other works investigated problems like leader election [16] and renaming [2].

False sharing [23, 11, 6] occurs when different objects, placed in the same block, are accessed by different processes, causing cache misses that would not occur if the objects were in separate blocks. Bolosky and Scott [6] introduced models for false sharing and compared how well they align with the intuitive understanding of this phenomenon. Their *interval definition* says that the cost of false sharing is the difference in performance between a policy that makes optimal placement decisions, but enforces consistency on a whole-block basis, and one that enforces consistency only for real conflicts between accesses. Our definitions capture this formally by the difference between the number of block RMRs and the number of RMRs, and allow us to prove the NP-hardness result conjectured in [6].

A large body of research studies memory locality for sequential computing. *Cache-conscious* algorithms take into account the structure of the cache, i.e., cache size, block size, placement of objects in blocks and other parameters, in order to minimize cache misses. Petrank and Rawitz [20] showed that the problem of partitioning data into the blocks of a single cache *of limited size* is NP-hard, and in fact, it is hard to approximate. Our NP-hardness proof for the CC model employs multiple processes, instead of bounding the size of the cache, and is achieved using traversals, rather than with an arbitrary access sequence as in their result. However, we do not show that approximating the optimal solution is hard; in fact, it is not, since the number of RMRs for the sequence divided by the size of the block is a lower bound on the number of block RMRs.

Lavaee [18] showed that the problem of *data packing*, using a fully associative, limited-size cache and for a single process, is NP-hard. By using multiple processes, our result is achieved with a more natural access sequence that does not require dummy objects in order to fill the cache and cause data to be evicted.

Afek et al. [1] introduce a memory allocation scheme that is cache index-aware, i.e., takes into account that objects are placed in a cache "row" that coincides with their cache *index*, which is a consecutive subset of the bits in their memory address.

Cache-oblivious algorithms [14, 22] optimize the object layout in the memory and their design is indifferent to cache parameters, such as size and block size. For example, a partitioning of tree nodes into blocks that reduces the number of blocks accessed is given by van Emde Boas trees [24]. Cache-oblivious algorithms were suggested for matrix transposition and FFT [14] for priority queues [4] and for sorting [14, 7, 12, 13]. All these algorithms are designed for a single process and do not take multi-threading into account.

More recent work attempts to provide cache-oblivious algorithms in multiprocessing environments. One such variant is the *parallel cache-oblivious* model [5]. In general, the number of cache misses for algorithms in this model is higher than in the regular cache-oblivious model, due to restrictions needed to minimize false sharing and memory imbalances between sub-tasks. However, for some tasks, the asymptotic bounds are not affected.

Multi-core oblivious cache-oblivious algorithms [10] are unaware of both the number of cores and the cache parameters. In these algorithms, different processes cooperate in order to complete a single task and data placement is designed to avoid cache misses as much as possible. Our work focuses on multi-process environment as well, but takes into account possible contention between noncooperative tasks run by different processes which may access the same object, for example, concurrent accesses to the same data structure. Our NP-hardness results hold even when the block size B is known and use only two processes. This makes them stronger and they also immediately apply to Multi-core oblivious cache-oblivious algorithms.

2 RMRs and Block RMRs in the Cache Coherence Model

We consider an asynchronous system in which a set of n processes, $P = \{p_1, \ldots, p_n\}$, execute concurrently and communicate by accessing a set O of shared objects. Each process has a local cache memory associated with it. Objects placed in the local memory can be easily retrieved, and the cost of doing so is negligible compared with the cost of fetching objects from another process's local cache, or from the main shared memory. A process may read an object's current value, or write a new value to an object.

Objects may be part of the same data structure, for example, vertexes of a graph. A common access pattern to such data structures is a *traversal*: a sequence of accesses by a single process to the vertexes of the graph, where each pair of consecutive accesses in the sequence π, either access the same vertex or adjacent vertexes.

The local memory is partitioned into *blocks* of *size* B, each of which can contain objects whose combined sizes is at most B. The local memory can hold an unbounded number of blocks and therefore, blocks are not evicted due to lack of space. A *B-block placement* of O is a partition of the objects in O into disjoint sets (*blocks*), $\tilde{O} = \{O_1, \ldots, O_\ell\}$, each containing objects with a combined size of at most B. We assume each object can be placed in a single block, and is not spread across blocks.[1]

If all objects have the same size, we will consider their size to be 1. In this case, a *B-block placement* of O is $\tilde{O} = \{O_1, \ldots, O_\ell\}$ such that for each block $|O_i| \leq B$.

An *access sequence* π is a sequence $(p_{i_1}, a_1, o_{j_1}), \ldots, (p_{i_m}, a_m, o_{j_m})$ such that $p_{i_h} \in P$, $a_h \in \{\text{read,write}\}$ and $o_{j_h} \in O$, for every h, $1 \leq h \leq m$.

There are two models for counting the number of *remote memory references* (*RMR*s) in an access sequence π, the *cache coherent* (*CC*) model and the *distributed shared memory* (*DSM*) model (see Figure 1). In this section, we concentrate on the CC model; the DSM model is discussed in Section 4.

In the CC model, a remote memory reference (RMR) is incurred either on the first access (whether it is a read or a write) to an object by a process or on the first access after a write to the same object by another process. Formally, for an access sequence $\pi = (p_{i_1}, a_1, o_{j_1}), \ldots, (p_{i_m}, a_m, o_{j_m})$

$$\begin{aligned} \#rmr^{\text{CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | (\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_h} \neq o_{j_k})) \text{ or} \\ (\exists k, 1 \leq k < h, (o_{j_k} = o_{j_h}, a_k = \text{write, and } p_{i_k} \neq p_{i_h}) \\ \text{and } (\forall \ell, k < \ell < h, (p_{i_h} = p_{i_\ell}) \implies (o_{j_h} \neq o_{j_\ell})))\}| \ . \end{aligned}$$

[1] Objects with size larger than B need more than one block. By placing an object in the minimal number of blocks that can contain it (i.e., all parts of the object except perhaps one are placed in a block alone), our results hold in this model, by ignoring the parts of the object that fill full blocks and taking into account only the part of the object that remains and does not fill a full block.

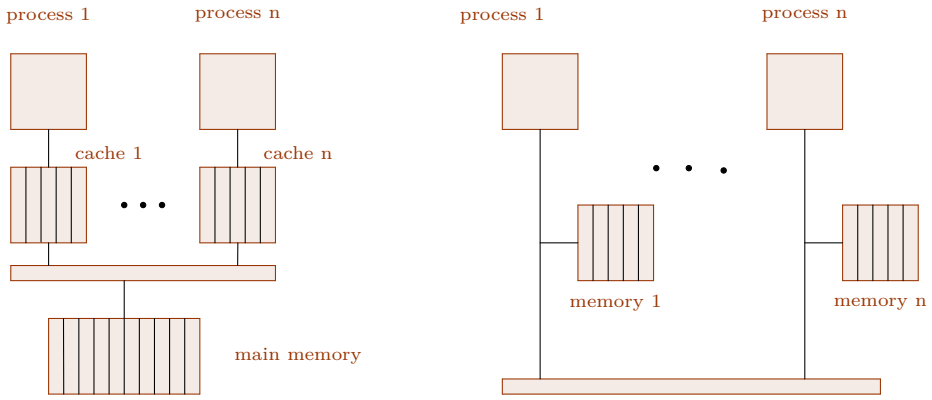


Figure 1 Models: Cache Coherence (left) and DSM (right).

Fix a B-block placement $\tilde{O} = \{O_1, \ldots, O_\ell\}$ for an access sequence π, as above. We count the number of *block* accesses in π that incur a cache miss, i.e., the first access (either a read or a write) to an object in a block, or an access after a write by another process to *some object in the same block*:

$$\begin{aligned} \#brmr^{\mathrm{CC}}(\pi, \tilde{O}) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | (o_{j_h} \in O_l) \text{ and} \\ ((\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_k} \notin O_l)) \text{ or} \\ (\exists k, 1 \leq k < h, (o_{j_k} \in O_l, a_k = \text{write, and } p_{i_k} \neq p_{i_h}) \\ \text{and } (\forall \ell, k < \ell < h, (p_{i_h} = p_{i_\ell}) \implies (o_{j_\ell} \notin O_l)))))\}| \,. \end{aligned}$$

3 Block RMRs in the CC model

In the cache coherence model, all objects are in the main memory before the access sequence is executed. Therefore, once the access sequence π is fixed, no optimization can reduce the number of RMRs and it can be computed by a sequential pass over the access sequence π, while tracking the last process that modified each object. Specifically, for every access, if there was no previous access in the sequence to the same object by the process or if the previous access was a modification by a different process, we increase the number of RMRs by one and update the latest process to access the object.

Note that when each object is placed in a separate block, that is, in the assignment $\tilde{O} = \{O_1, \ldots, O_n\}$, with $|O_i| = 1$, for every i, $1 \leq i \leq n$, we have:

$$\#brmr^{\mathrm{CC}}(\pi, \tilde{O}) = \#rmr^{\mathrm{CC}}(\pi).$$

Therefore, $\#rmr^{\mathrm{CC}}(\pi)$ is an upper bound on the minimal number of block RMRs for the sequence π.

Given a B-block placement of O, $\tilde{O} = \{O_1, \ldots, O_n\}$, let $\#rmr^{\mathrm{CC}}(o_i, \pi)$ be the number of accesses to the object o_i that incur an RMR, i.e., an access to o_i that caused the block containing o_i to be brought to the process's local memory. For every access to o_i that incurs an RMR, there is a block RMR that brings o_i to the accessing process, incurred by either the access itself or a previous access to another object in the block. Therefore, the number

of block RMRs for objects in O_j is at least

$$\max_{o_i \in O_j} (\#rmr^{\mathrm{CC}}(o_i, \pi)).$$

Since each block contains at most B objects, the number of block RMRs is at least

$$\frac{\Sigma_{o_i \in O_j} (\#rmr^{\mathrm{CC}}(o_i, \pi))}{B}.$$

The overall number of block RMRs is the sum of block RMRs for the objects in each block O_j and therefore, it is at least

$$\Sigma_{O_j \in \tilde{O}} \frac{\Sigma_{o_i \in O_j} (\#rmr^{\mathrm{CC}}(o_i, \pi))}{B} = \frac{\#rmr^{\mathrm{CC}}(\pi)}{B}.$$

Therefore, for every access sequence π,

$$\frac{\#rmr^{\mathrm{CC}}(\pi)}{B} \leq \min_{\tilde{O}} \#brmr^{\mathrm{CC}}(\pi, \tilde{O}) \leq \#rmr^{\mathrm{CC}}(\pi).$$

We consider the question of optimally placing objects into blocks, in order to minimize the number of block RMRs. We prove that for $B > 2$, the problem is NP-hard, even if the sequence of accesses is known in advance, by showing a polynomial-time reduction from the *graph partitioning problem* [17]. But first, we prove that there is a polynomial-time algorithm for this problem when $B = 2$.

3.1 A Polynomial Algorithm for 2-Block Placement

A 2-block placement can be found with an algorithm for finding a maximum weighted matching for a graph, which can be done in $O(|V|^2|E|)$ using linear programming [21].

▶ **Definition 1** (Maximum Weighted Matching)**.**
Input: Undirected graph $G = (V, E)$, weights $w(e) \in Q$ for each edge $e \in E$.
Question: Which matching, i.e., a set of pairwise non-adjacent edges, has maximum weight (the sum of the weights of the edges in the matching)?

Given an input to the 2-block placement problem, we take the complete graph $G = (V, E)$ with a vertex $v_i \in V$ for every object $o_i \in O$ and an edge between every pair of vertexes.

The edge weights are calculated according to the access sequence π. For each access by process p_i to an object o_1 and every object o_2, we determine whether the access would incur an RMR if o_1 and o_2 are placed in the same block. To do this, we check if o_2 was previously accessed by p_i. If there is a previous access to o_2 by p_i, we check whether an access by another process in between could invalidate the block in the local memory and decrease the weight of the edge between o_1 and o_2. Otherwise, we check which previous access could save an RMR if o_1 and o_2 are placed in the same block and increase the weight of the edge between o_1 and o_2. For example, Figure 2(left) shows the final weights for the next access sequence on four objects A, B, C, D:

$$(p_1, \text{write}, A)(p_1, \text{write}, B)(p_2, \text{write}, C)(p_3, \text{write}, D)(p_2, \text{write}, C)$$

The weight of (A, B) is 1 because placing them in the same block would save a block RMR on the access to B. The weight of (C, D) is -1 because placing them in the same block would incur an extra block RMR on the second access to C (which does not happen if C is placed in a singleton block).

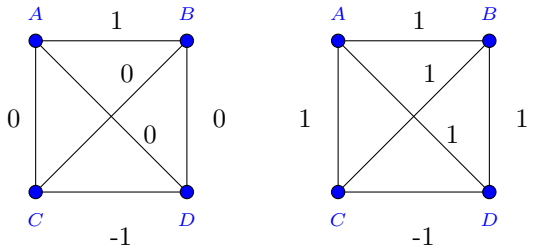


Figure 2 Graph with original weights (left) and after incorrect weight adjustment (right).

After the weights are calculated, we find a maximum matching in the weighted graph. A matching and a 2-block placement naturally correspond to each other: the endpoints of edges in the matching represent disjoint pairs of objects, and each pair can be placed in a single block in the memory. The remaining objects are placed in singleton blocks.

The weights are calculated according to the access sequence π (Algorithm 1). We initialize the weight for each edge to zero. Next we go over the access sequence, and for each access to an object, check which previous accesses to other objects could either increase or decrease the number of RMRs if they are put in the same block as the current object. For each access $(p_{i_h}, a_h, o_{j_h}) \in \pi$, let $(p_{i_k}, a_k, o_{j_h}) \in \pi$ be the previous access to the same object in π. If no such access exists, for example, if (p_{i_h}, a_h, o_{j_h}) is the first access to the object, we look for previous accesses by the same process. For every object o, whose most recent access (prior to the h-th access) is by p_{i_h}, we increase the weight of (o, o_{j_h}) by 1, since placing them in the same block will avoid a block RMR on the access (p_{i_h}, a_h, o_{j_h}).

Now we assume (p_j, a_j, o_j) exists. If $p_j = p_i$, then for each object o', if there is a process $p' \neq p_i$ such that there is an index k such that $j < k < i$ and $p_k = p'$ and $o_k = o'$, and this is the most recent write to o' prior to the i-th access, we decrease the weight of the edge (o', o_i) by 1. For example, for three processes, objects $O = \{A, B, C, D\}$, and the sequence:

$$(p_1, \text{write}, A), (p_2, \text{write}, B), (p_1, \text{write}, C), (p_3, \text{write}, B), (p_1, \text{write}, A),$$

the weight of the edge (A, B) will be decreased by 1, since there is a write to B by a process other than p_1 between the two accesses to A. Intuitively, if A and B are placed in the same block then we will incur an extra RMR for this part of the sequence, which would not be incurred had they been placed in different blocks.

If, on the other hand, $p_j \neq p_i$, then for each object o' such that there is an index k, $j < k < i$, $p_k = p_i$ and $o_k = o'$, and this is the most recent write to o' prior to the i-th access, we increase the weight of the edge (o', o_i) by 1. For example, for the sequence:

$$(p_4, \text{write}, D), (p_2, \text{write}, B), (p_1, \text{write}, C), (p_2, \text{write}, B), (p_2, \text{write}, D)$$

the weight of the edge (B, D) will be increased by 1, since there is a write to B by process p_2 between the two writes to D by p_2. Intuitively, if D and B are placed in the same block then we will incur one less RMR for this part of the sequence than would have been incurred had they been placed in different blocks.

Figure 3 shows the final weights of the graph for the sequence that is the concatenation of the two previous sequences (all actions are writes):

$$(p_1, A), (p_2, B), (p_1, C), (p_3, B), (p_1, A), (p_4, D), (p_2, B), (p_1, C), (p_2, B), (p_2, D)$$

Algorithm 1 Sequential pseudocode for calculating the weights.

```
    calc_weights(G = (V, E), w, acc)
 1: init w to zeros
    //latest and latest_write are arrays of size |V| representing,
    //respectively, the most recent access and write to an object
 2: init latest to nulls
 3: init latest_write
 4: for i = 1...|acc|:
 5:       (curr_proc, curr_act, curr_obj) = acc[i]
 6:       prev_acc = latest[curr_obj]
 7:       if (prev_acc is null):
              // Find which objects were previously accessed by the
              // same process, and increase the corresponding weight
 8:           for obj = 1...|V|:
 9:               if (obj ≠ curr_obj and latest[obj] ≠ null):
10:                   (proc, temp1, temp2) = acc[latest[obj]]
11:                   if (proc == curr_proc):
12:                       w((obj, curr_obj))+ = 1
13:      else:
14:          prev_proc = acc[prev_acc][0]
15:          for obj = 1...|V|:
                 // Check if the obj was accessed between the previous access
                 // to curr_obj and the current access
16:              if(obj ≠ curr_obj and latest[obj] ≠ null
                 and latest[obj] > prev_acc):
17:                  (proc, temp1, temp2) = acc[latest[obj]]
                     // Putting obj and current_obj together can save a block RMR
18:                  if (prev_proc ≠ curr_proc
                       and proc = curr_proc):
19:                      w((obj, curr_obj))+ = 1
                     // Putting obj and current_obj together can incur an extra block RMR
20:                  else if (latest_write[obj] ≠ null
                       and latest_write[obj] > prev_acc
                       and prev_proc = curr_proc
                       and acc[latest_write[obj]][0] ≠ curr_proc):
21:                      w((obj, curr_obj))− = 1
      // Update latest and latest_write according to the access
22:   latest[curr_obj] = i
23:   if curr_act is a write:
24:       latest_write[curr_obj] = i
```

▶ **Lemma 2.** *If the matching $M_{\tilde{O}} \subseteq E$ corresponds to a 2-block placement $\tilde{O} = \{O_1, \ldots, O_\ell\}$, then $\#rmr^{\mathrm{CC}}(\pi) - \Sigma_{e \in M_{\tilde{O}}} w(e) = \#brmr^{\mathrm{CC}}(\pi, \tilde{O})$.*

Proof. Consider some block $O_j \in \tilde{O}$. If $O_j = \{a\}$ is a singleton block, then the number of RMRs incurred by access to a is $\#rmr^{\mathrm{CC}}(a)$, since accesses to other objects do not cause it to move to a different cache. On the other hand, if O_j is mapped to some edge $e_{O_j} \in M_{\tilde{O}}$ it contains two objects, a and b. We count the block RMRs incurred by accesses to a and b. For every access to a (and similarly b) there are three cases:

1. The access is counted as an RMR in $\#rmr^{\mathrm{CC}}(\pi)$ and a block RMR in $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$.
2. The access is counted as an RMR in $\#rmr^{\mathrm{CC}}(\pi)$, but not in $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$. This happens only if the access to a was preceded by an access to b, that brought O_j to the accessing process' cache. This can happen in either the first access to a or in subsequent ones. We note that for each such occurrence we increased $w(e_{O_j})$ by 1.

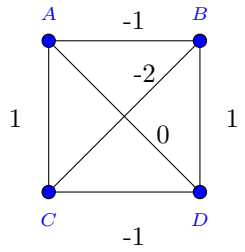


Figure 3 Graph after the weights are calculated.

3. The access is counted as an RMR $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$, but not in $\#rmr^{\mathrm{CC}}(\pi)$. This happens only if the access to a was preceded by an access to b, which moved the block O_j from the process' cache. We note that for each such occurrence we decreased $w(e_{O_j})$ by 1.

Therefore, the number of block RMRs incurred by a and b is exactly

$$\#rmr^{\mathrm{CC}}(a) + \#rmr^{\mathrm{CC}}(b) - w(e_{O_j}).$$

Summing over all blocks, we get:

$$\#brmr^{\mathrm{CC}}(\pi, \tilde{O}) = \Sigma_{O_j \in \tilde{O}, O_j = \{a\}}(\#rmr^{\mathrm{CC}}(a)) + \Sigma_{O_j \in \tilde{O}, O_j = \{a,b\}}(\#rmr^{\mathrm{CC}}(a) + \#rmr^{\mathrm{CC}}(b) - w(e_{O_j})).$$

And thus:

$$\#rmr^{\mathrm{CC}}(\pi) - \Sigma_{e \in M_{\tilde{O}}} w(e) = \#brmr^{\mathrm{CC}}(\pi, \tilde{O}).$$ ◀

Given this lemma, it is easy to see that if $\Sigma_{e \in M_{\tilde{O}}} w(e)$ is maximized, then our target function $\#brmr^{\mathrm{CC}}(\pi, \tilde{O})$ is minimized. Finding the maximum for $\Sigma_{e \in M_{\tilde{O}}} w(e)$ is finding a maximum weight matching in a weighted graph, which can be solved in $O(|V|^2 \cdot |E|)$ steps [21]. Since $|V| = |O|$ and the graph is complete, this is in $O(|O|^3)$. The total complexity of the algorithm depends on the calculation of edge weights. Using a straightforward approach, for every access we must go over all previous accesses and update an edge to every object, resulting in an $O(|\pi|^2 + |\pi| \cdot |O| + |O|^3)$ time complexity. This can be improved by using a hash table to remember the last read and write for each object while going over the access sequence. This alleviates the need to go over all previous accesses; instead, we go over the objects and find which were accessed in the relevant part of the sequence. This results in $O(|\pi| \cdot |O| + |O|^3)$ time complexity.

The algorithm does not guarantee that between two solutions with the same minimal number of block RMRs, it chooses the one with the smallest number of blocks. This can be done by maximizing the number of blocks containing two objects, or equivalently, maximizing the number of edges in the matching. Therefore, we look for a maximum-weight matching with as many edges as possible.

We note that any such solution cannot contain negative-weighted edges, since removing them would increase the total weight. Note also that removing 0-weighted edges still leaves a maximum-weight matching. Therefore, the maximum-weight matching with the maximal number of edges is comprised of a maximum-weight matching with as many 0-weighted edges added to it as possible. By adding a small positive weight to the 0-weighted edges before running the maximum-weight matching algorithm, we can ensure that as many such edges

as possible will be added to the solution, since adding them will only increase the total weight. This weight must be the same for all edges, since we do not prefer one 0-weighted edge over the other. In addition, the weight must be small enough to ensure that the new solution contains some original maximum-weight matching as a subset. For example, every maximum-weight matching for the graph of the sequence:

$$(1, A)(1, B)(2, C)(3, D)(2, C)$$

must contain the edge (A, B), since it is the only edge with positive weight.

If the weight added to 0-weighted edges is 1, as shown in Figure 2(right), then a maximum-weight matching on this graph is either the edges $(A, D), (B, C)$ or $(A, C), (B, D)$, neither of which contain the edge (A, B). However, if we give all the 0-weighted edges a weight $< \frac{1}{|E|}$ (for example $\frac{1}{|V|^2}$), the total weight of all the 0-weighted edges is smaller than 1. Since all the positive-weighted edges weigh at least 1, any combination of originally 0-weighted edges weighs less than the weight of those edges. Therefore, the solution found by the algorithm must contain an original maximum-weight matching as a subset, otherwise, such a matching will produce a higher total weight, contradicting the algorithm's optimality in finding a maximum-weight matching. This implies the next theorem:

▶ **Theorem 3.** *In the CC model, there is a polynomial algorithm for finding a 2-block placement, with the minimal number of block RMRs for a given access sequence, while minimizing the total number of blocks used.*

3.2 Hardness Proof for B-Block placement, $B > 2$

We now prove a hardness result for B-block placement with $B > 2$, by showing a polynomial-time reduction from the graph partitioning problem, known to be *NP-complete*, even for a fixed $K \geq 3$ and even if all vertex and edge weights are 1 [17].

▶ **Definition 4** (Graph Partitioning)**.**

Input: Undirected graph $G = (V, E)$, weights $w(v) \in Z^+$ for each vertex $v \in V$ and $l(e) \in Z^+$ for each edge $e \in E$, positive integers K and J.

Question: Is there a partition of V into disjoint sets $V_1, \ldots, V_m$, such that $\Sigma_{v \in V_i} w(v) \leq K$ for $1 \leq i \leq m$ and such that if $E' \subseteq E$ is the set of edges that have their two endpoints in two different sets V_i then $\Sigma_{e \in E'} l(e) \leq J$?

We redefine B-block placement as a decision problem.

▶ **Definition 5** (B-Block Placement Decision)**.**

Input: Two positive integers B and R, a set of n processes $P = \{p_1, \ldots, p_n\}$, a set of memory objects O, a sequence of accesses $\pi = (p_1, o_1), \ldots, (p_m, o_m)$ such that for every i, $1 \leq i \leq m$, $p_i \in P$ and $o_i \in O$.

Question: Is there a partition of the objects in O into disjoint sets $\tilde{O} = \{O_1, \ldots, O_k\}$ such that $|O_i| \leq B$ and $\#brmr^{\mathrm{CC}}(\pi, \tilde{O}) < R$?

Given that it is easy to calculate the number of RMRs, and it is an upper bound on the number of block RMRs, a polynomial time algorithm for the decision problem can be used in conjunction with a binary search to solve the optimization problem.[2] Therefore, if we prove that the decision problem is NP-hard, the minimization problem is also NP-hard.

[2] That is, what is the minimal number of block RMRs for the sequence π. An optimal placement of objects to blocks is not found in this way.

We will use the input graph for which partitioning must be found as the underlying data structure that two processes access simultaneously. Each process performs a traversal and accesses the edges in DFS order. Processes access the edges in round-robin order, each accessing the endpoints of an edge one after the other, before control is passed to the next process. This access pattern ensures that putting the two endpoints of an edge in the same block results in fewer block RMRs than if they are in different blocks. Therefore, given a placement with fewer block RMRs, the partition into blocks induces a partition of the graph into disjoint sets of vertexes, which gives a good solution to the graph partitioning problem. Conversely, we prove that if there is no such placement, then there is no valid solution to the graph partitioning problem.

In more detail, the input is an undirected graph $G = (V, E)$, in which all edge and vertex weights are 1, and positive integers $K \geq 3$ and J.

We take two processes p_1 and p_2 and an object o for each vertex $v \in V$. For each edge $e \in E, e = (v_i, v_j)$ and process p, we define a subsequence

$$\pi_{(p,e)} = (p, \text{read}, o_i)(p, \text{write}, o_i), (p, \text{read}, o_j)(p, \text{write}, o_j).$$

Let $\pi_e = \pi_{(p_1,e)}, \pi_{(p_2,e)}$. Consider the following traversal in DFS order of the edges of the graph: The traversal starts at an arbitrary vertex. When the traversal reaches a node, it either immediately retreats through the same edge, if the node was already visited, or it continues to visit the node's neighbors, and then retreats through the same edge. Therefore, each edge is visited twice during the traversal, and neighboring vertexes are accessed one after the other. Let $e_{i_1}, \ldots, e_{i_{2|E|}}$ be the sequence of edges in this traversal. We define $\pi = \odot_{1 \leq j \leq 2|E|} \pi_{e_{i_j}}$. For each process p, the sequence of accesses is $\odot_{1 \leq j \leq 2|E|} \pi_{(p,e_{i_j})}$, which is a traversal due to the choice of the order of the edges. The length of the access sequence π is in $O(|E)|$, since it is a concatenation of $2|E|$ constant size sequences, and therefore, the reduction is polynomial in the size of the input. Let $B = K$ and $R = 4(|E| + J)$.

▶ **Lemma 6.** *There is a B-block placement of O for sequence π with R or fewer block RMRs if and only if there is a partitioning of the graph G under the K and J weight sum constraints.*

Proof. Given a graph and a partition $V_1, \ldots, V_m$ of the vertexes such that $\Sigma_{v \in V_i} w(v) \leq K$, we define a B-block placement for $B = K$ where O_i contains all the objects that correspond to the vertexes in V_i. Since all weights are 1, the number of vertexes in V_i is at most K, and so is the number of objects in O_i. Similarly, a B-block placement of the objects induces a partition of the corresponding vertexes that satisfies the $K = B$ constraint on the graph.

Let E' be the set of edges with endpoints in different sets V_i. We argue that the number of block RMRs for the sequence π is $4(|E| + |E'|)$. After each subsequence π_e, each block is either still in the main memory or in the cache of process p_2, since p_2 modifies each block after p_1 modifies it. Therefore, if $e = (v_i, v_j)$ and o_i and o_j, the corresponding objects, are in the same block, then the number of block RMRs for π_e is exactly 2, since each process must bring the block into its cache (including the first process) in order to access o_i, and then proceed to access o_j. Otherwise, if o_i and o_j are in different blocks, the number of block RMRs for π_e is exactly 4, since each process must access both blocks in turn. Therefore, since every edge is traversed twice, the total number of block RMRs is $4|E \backslash E'| + 2 \cdot 4|E'| = 4(|E| + |E'|)$.

It remains to show that a solution to the B-block placement problem implies a solution to the graph partitioning problem. If there is no B-block placement for $B = K$ and the sequence π with $R = 4(|E| + J)$ or fewer block RMRs, then for every placement and corresponding graph partitioning, the number of block RMRs is $4(|E| + |E'|) > R = 4(|E| + J)$ and therefore, $|E'| > J$. Hence, there is no graph partitioning such that the size of every group is K or less and the number of edges between different groups is J or less.

Table 1 Remote memory references in variants of the DSM model.

	Reads	Writes	Invalidation
DSM - With Coherence	Free as long as data in local memory is valid. Incurs an RMR, otherwise.	Free if data in local memory is valid, and incurs an RMR, otherwise. Causes invalidation of object copies in other local memories.	Negligible
DSM - No Coherence	Reads by object creator are free. Other reads incur an RMR.	Writes to an object by a non creator process incur an RMR. Writes by the owner are free.	Not supported

Conversely, if there is a B-block placement for $B = K$ and the sequence π, with fewer than $R = 4(|E| + J)$ RMRs, then $4(|E| + |E'|) \leq R = 4(|E| + J)$, and $|E'| \leq J$. Therefore, the corresponding partitioning of G satisfies the constraints on K and J. ◀

This proves the next theorem:

▶ **Theorem 7.** *In the CC model, the B-block placement decision problem, for $B \geq 3$, is NP-hard.*

4 Block RMRs in the DSM Model

In the DSM model, an access to another process' local memory incurs an RMR, depending on the specific characteristics of the implementation. Some DSM systems incorporate some form of *cache coherence*, i.e., a mechanism that ensures processes only read or write valid data, and not data that was already overwritten by another process. We consider two variants of the DSM model, depending on whether invalidating an object in a remote memory is possible, and what is the cost of doing so (see Table 1).[3]

Each object o is created at a *creator* process, denoted *creator*(o), which is the owner of the local memory in which o originally resides. The *creator*(o) process is chosen before the first access to o. An object is valid when created, and therefore, accesses to the object are free as long as they are just by the creator process. In all variants of the DSM model, we may choose *creator*(o) according to whatever information we have on the access sequence.

If coherence is supported and invalidation cost is negligible (for example, see [19]), then each read or write to an object not in the local memory causes the object to be copied to the local memory, incurring an RMR. If the action is a write then other copies of the object are invalidated, and the cost of doing so is negligible. Given an access sequence $\pi = (p_{i_1}, a_1, o_{j_1}), \ldots, (p_{i_m}, a_m, o_{j_m})$, we formally define:

$$\begin{aligned}\#rmr^{\mathrm{DSM-CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | ((p_{i_h} \neq \mathit{creator}(o_{j_h}))& \\ \text{and } (\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_h} \neq o_{j_k})))& \\ \text{or } (\exists k, 1 \leq k < h, (o_{j_k} = o_{j_h}, a_k = \text{write, and } p_{i_k} \neq p_{i_h})& \\ \text{and } (\forall \ell, k < \ell < i, (p_{i_h} = p_{i_\ell}) \implies (o_{j_h} \neq o_{j_\ell})))\}| \,.&\end{aligned}$$

If coherence and invalidation are not supported, then every access, whether a write or a read, to an object by a process other than its creator, incurs an RMR. Accesses to an object

[3] In the full version of the paper we also consider a third variant, in which invalidation is supported, but its cost is similar to an RMR.

Table 2 *Block* remote memory references in variants of the DSM model.

	Reads	Writes	Invalidation
DSM - With Coherence	Free as long as data in local memory is valid. Incurs an RMR, otherwise.	Free if data in local memory is valid, and incurs an RMR, otherwise. Causes invalidation of other copies in other local memories.	Negligible
DSM - No Coherence	Reads by the object's creator process are free. Other reads incur an RMR.	Writes by owner process are free. Other writes incur an RMR.	Not supported

by its creator are free. That is:

$$\#rmr^{\mathrm{DSM-No-CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | p_{i_h} \neq \mathit{creator}(o_{j_h})\}| \ .$$

To define block RMRs in these variants of the DSM model, consider a B-block placement $\tilde{O} = \{O_1, \ldots, O_\ell\}$. Each block O_i is associated with a creator process $\mathit{creator}(O_i)$, which creates all objects in O_i. The RMRs incurred for reads and writes is shown in Table 2, and defined next.

If coherence is supported with negligible invalidation cost, then each access to an object not in the local memory incurs an RMR; it causes a copy of the accessed block to appear in the local memory. If the access is a write then all other copies of the block are invalidated. Accesses to objects whose blocks are in the local memory are free. Formally:

$$\begin{aligned}\#brmr^{\mathrm{DSM-CC}}(\pi, \tilde{O}) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | (o_{j_h} \in O_l) \text{ and } \\ (((p_{i_h} \neq \mathit{creator}(O_l)) \text{ and } \\ (\forall k, 1 \leq k < h, (p_{i_h} = p_{i_k}) \implies (o_{j_k} \notin O_l))) \text{ or } \\ (\exists k, 1 \leq k < h, (o_{j_k} \in O_l, a_k = \text{write, and } p_{i_k} \neq p_{i_h}) \\ \text{and } (\forall \ell, k < \ell < h, (p_{i_h} = p_{i_\ell}) \implies (o_{j_\ell} \notin O_l))))\}| \ .\end{aligned}$$

When coherence is not supported, each access to a block by a process other than its creator incurs an RMR; accesses by the creator are free. Formally:

$$\#brmr^{\mathrm{DSM-no-CC}}(\pi) = |\{(p_{i_h}, a_h, o_{j_h}) \in \pi | o_{j_h} \in O_\ell \text{ and } p_{i_h} \neq \mathit{creator}(O_\ell)\}| \ .$$

The DSM model with coherence is similar to the CC model, except for the first access to every block: If the first access to an object o is done by process $p \neq \mathit{creator}(o)$, then the step incurs an RMR; otherwise, no RMR is incurred. Therefore the number of RMRs depends on the choice of $\mathit{creator}(o)$ for these accesses, while other accesses are not affected.

We explain how the results for the CC model can be extended to this model. The polynomial algorithm for blocks with $B = 2$ is adapted by changing the edge weights to account for the objects being created in the process' local memory instead of the main memory. The NP-hardness result with $B > 2$ can also be adapted by a slight change in the access sequence and the way edge weights are calculated (see Section B).

In the DSM model without coherence, the number of RMRs is minimized when each object is created at the process that accesses it the most. Furthermore, it can be shown that the minimal number of RMRs does not depend on the partitioning into blocks, just on the choice of creator processes. This means that a simple greedy algorithm can be used to pick the creator process for each object.

5 Conclusions and Future Research Directions

This paper introduces a framework for studying the cost of accessing remote memory (whether shared memory or data stored at another process), which takes into account the fact that shared objects can be placed and moved together in larger blocks. We introduce a formal complexity measure, called *block RMR*s, for both the CC and the DSM models. Our main result shows that it is NP-hard to place objects into blocks in a way that minimizes the number of block RMRs, even for a fixed access sequence. The result holds for both the CC model and the DSM model with coherence and negligible invalidation cost, when a block can contain three objects or more.

In the common situation, however, the access sequence is not known in advance. Instead, we know it is from a *family* of sequences, typically, those generated by a particular concurrent algorithm, for example, interleavings of (partial) traversals or searches by a subset of the processes. It would be interesting to find block placement methodologies for such families, in a way that exploits the benefits of moving several objects together. Taking this a step further, it is interesting to look at probabilistic models where the access sequences are chosen from a family of sequences with a known distribution. The goal is to choose a B-block placement such that the *expected* number of block RMRs is minimized. In the DSM model without coherence, these problems may have simple solutions in the form of choosing the object's creator to be the process that is expected to access it the most.

It would also be interesting to study the effects of bounding the local memory size, so it can hold a bounded number of blocks, and limiting the cache associativity. It is likely that in general, the problem is *NP-hard* due to our results as well as [20, 18].

Acknowledgements. We would like to thank Youla Fatourou, Danny Hendler, Erez Petrank and the referees for helpful comments and useful discussions.

References

1 Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. *SIGPLAN Not.*, 46(11):55–64, 2011. doi:10.1145/2076022.1993486.

2 Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 718–727, 2011.

3 James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003. doi:10.1007/s00446-003-0088-6.

4 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC, pages 268–276, 2002. doi:10.1145/509907.509950.

5 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 355–366, 2011. doi:10.1145/1989493.1989553.

6 William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, 1993. URL: http://dl.acm.org/citation.cfm?id=1295480.1295483.

7 Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming*, ICALP, pages 576–588, 2005. doi:10.1007/11523468_47.

8 Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, 1998. doi:10.1145/291006.291036.

9 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI, pages 1–12, 1999.

10 Rezaul Alam Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

11 Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *Proceedings of the International Conference on Parallel Processing, ICPP. Volume I: Architecture/Hardware*, pages 377–381, 1991.

12 Rolf Fagerberg, Anna Pagh, and Rasmus Pagh. External string sorting: Faster and cache-oblivious. In *Proceedings of the 23rd Annual Conference on Theoretical Aspects of Computer Science*, STACS, pages 68–79, 2006. doi:10.1007/11672142_4.

13 Arash Farzan, Paolo Ferragina, Gianni Franceschini, and J. Ian Munro. Cache-oblivious comparison-based algorithms on multisets. In *Proceedings of the 13th Annual European Conference on Algorithms*, ESA, pages 305–316, 2005. doi:10.1007/11561071_29.

14 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

15 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

16 Wojciech Golab, Danny Hendler, and Philipp Woelfel. An $O(1)$ RMRs leader election algorithm. *SIAM J. Comput.*, 39(7):2726–2760, 2010.

17 Laurent Hyafil and Ronald L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical Report Rapport de Recherche no. 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, October 1973.

18 Rahman Lavaee. The hardness of data packing. In *Proceedings of the 43rd Annual ACM Symposium on Principles of Programming Languages*, POPL, pages 232–242, 2016. doi:10.1145/2837614.2837669.

19 Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, aug 1991. doi:10.1109/2.84877.

20 Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. *SIGPLAN Not.*, 37(1):101–112, 2002. doi:10.1145/565816.503283.

21 M.D. Plummer and L. Lovász. *Matching Theory*. North-Holland Mathematics Studies. Elsevier Science, 1986. URL: https://books.google.co.il/books?id=mycZP-J344wC.

22 Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massacusetts Institute if Technology, Cambridge, MA, 7 1999.

23 Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

24 P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Symposium on Foundations of Computer Science*, pages 75–84, 1975.

25 Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995. doi:10.1007/BF01784242.

A B-Block Placement for Objects with Varying Sizes

We prove that if objects have varying sizes, then the B-block placement problem is *NP-hard* even for a simple traversal on a tree. Assume that each object o_i has a size $a_i \in Z^+$, $a_i \leq B$, and each object is placed in a single block (and not across more than one block). A B-block placement is a partition of the objects in O into disjoint sets $\tilde{O} = \{O_1, \ldots, O_n\}$ such that, for every $1 \leq j < n$, $\Sigma_{o_i \in O_j} a_i \leq B$.

For both the CC and the DSM models, there is a reduction from the *bin packing* problem, a well-known NP-complete problem [15].

▶ **Definition 8** (Bin Packing).
Input: An integer R (the bin size), and ℓ items with sizes $a_1, \ldots, a_\ell$.
Question: What is the minimum number M such that there is a partition of the ℓ items into disjoint sets $S_1, \ldots, S_M$ such that $\Sigma_{i \in S_j} a_i \leq R$ for every j, $1 \leq j \leq M$?

Fix R to be the block size B, the number of objects to be ℓ and the size of an object o_i be a_i. We define a tree $T = (V, E)$ whose vertexes are v_i for each object o_i, and whose edges are $E = \{(v_i, v_{i+1}) | 1 \leq i < \ell\}$. For processes $p_1, p_2 \in P$, consider the access sequence $\pi = (p_1, \text{write}, o_1), \ldots, (p_1, \text{write}, o_\ell), (p_2, \text{write}, o_1), \ldots, (p_2, \text{write}, o_\ell)$, in which p_1 and p_2 write to each object once.

An optimal algorithm for B-block placement packs the objects into the smallest number of blocks possible: Since the size of the local memory is unlimited, any block read into the local memory remains there until it is required by a different process in the CC and DSM with coherence models.

Therefore, the number of block RMRs is equal to twice the number of blocks in which the objects reside in the CC model and DSM with coherence models.

Thus, an algorithm that finds the optimal solution to the B-block placement problem also finds a partitioning of objects of sizes $a_1, \ldots, a_\ell$ into blocks of size $B = R$, which minimizes the number of blocks used. This yields an optimal solution to the bin packing problem. Since bin packing is NP-complete [15], so is B-blocking for objects of varying sizes.

B NP-Hardness for DSM with Coherence and $B > 2$

We explain how to adapt the NP-hardness result for $B \geq 3$. Given the parameters to the graph partitioning problem, the parameters to the B-Block mapping are very similar: The input is an undirected graph $G = (V, E)$, weights $w(v) \in Z^+$ for each vertex $v \in V$ and $l(e) \in Z^+$ for each edge $e \in E$, and positive integers K and J. We assume that all weights are 1, and $K \geq 3$.

The access sequence is somewhat different: We take two processes p_1 and p_2, and an object o for each vertex $v \in V$. For each edge $e \in E, e = (v_i, v_j)$ and process p, we define a subsequence

$$\pi_{(p,e)} = (p, \text{read}, o_i)(p, \text{write}, o_i), (p, \text{read}, o_j)(p, \text{write}, o_j)).$$

Let $\pi_e = (\pi_{(p_1,e)}, \pi_{(p_2,e)})^{|E|}$. We define $\pi = \odot_{e \in E} \pi_e$. The length of π is $O(|E|^2)$ and therefore, the reduction is in polynomial time. Let $B = K$ and $R = 4|E|(|E| + J) - |V|$.

Given the part of the access sequence corresponding to an edge $e = (v_i, v_j)$,

$$\pi_e = ((p_1, \text{write}, o_i), (p_1, \text{write}, o_j)), (p_2, \text{write}, o_i), (p_2, \text{write}, o_j)^{|E|},$$

the number of block RMRs is as follows: If both endpoints of e are in the same block, then every access by p_1, except perhaps the first one, incurs a block RMR. This is because at

the end of $\pi_{(p_2,e)} = (p_2, \text{write}, o_i), (p_2, \text{write}, o_j)$, the block is in p_2's local memory. The total number of block RMRs is $2|E|$, from which we reduce 1 if this is the first access to the block containing o_i and o_j.

On the other hand, if the endpoints of e are in different blocks, then every access by either p_1 or p_2 must move a block into its local memory, except perhaps the first two accesses of p_1. The total number of block RMRs is $4|E|$, minus 1 or 2, depending on whether or not this sequence accessed blocks for the first time.

Since at the end of each subsequence π_e all blocks that were already accessed are in the local memory of process p_2, if $|E'|$ is the number of edges with endpoints in different blocks, the total number of RMRs is at most

$$2|E| \cdot 2|E \backslash E'| + 4|E| \cdot 2|E'| = 4|E|(|E| + |E'|)$$

and at least

$$2|E| \cdot 2|E \backslash E'| + 4|E| \cdot 2|E'| - |V| = 4|E|(|E| + |E'|) - |V|,$$

where $|V|$ is the maximum possible number of blocks.

If there is no B-block ent for $B = K$ and the sequence π with $R = 4|E|(|E| + J) - |V|$ or fewer block RMRs exists, then for every placement and corresponding graph partitioning the number of block RMRs is at least

$$4|E|(|E| + |E'|) - |V| > R = 4|E|(|E| + J) - |V|$$

and therefore $|E'| > J$. Hence, there is no graph partitioning such that the size of every group is K or less and the number of edges between different groups is J or less.

Conversely, if there is a B-block placement for $B = K$ with S blocks, and the sequence π, induces fewer than $R = 4|E|(|E| + J)$ block RMRs, then

$$4|E|(|E| + |E'|) - S \leq R = 4|E|(|E| + J) - |V|,$$

and therefore

$$4|E| \cdot |E'| \leq 4|E| \cdot J - (|V| - S).$$

Since $|V| - S > 0$, we have $|E'| \leq J$. Therefore, the corresponding partitioning of G holds under the constraints on K and J.

This proves the following theorem:

▶ **Theorem 9.** *In the DSM model with cache coherence and negligible invalidation, the B-block placement decision problem is NP-hard, for $B \geq 3$.*

Constant-Space Population Protocols for Uniform Bipartition

Hiroto Yasumi[1], Fukuhito Ooshita[2], Ken'ichi Yamaguchi[3], and Michiko Inoue[4]

1 **College of Information Science, National Institute of Technology, Nara College, Nara, Japan**
a0858@stdmail.nara-k.ac.jp
2 **Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan**
f-oosita@is.naist.jp
3 **College of Information Science, National Institute of Technology, Nara College, Nara, Japan**
yamaguti@info.nara-k.ac.jp
4 **Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan**
kounoe@is.naist.jp

Abstract

In this paper, we consider a uniform bipartition problem in a population protocol model. The goal of the uniform bipartition problem is to divide a population into two groups of the same size. We study the problem under various assumptions: 1) a population with or without a base station, 2) weak or global fairness, 3) symmetric or asymmetric protocols, and 4) designated or arbitrary initial states. As a result, we completely clarify constant-space solvability of the uniform bipartition problem and, if solvable, propose space-optimal protocols.

1998 ACM Subject Classification C.2.4 Distributed Systems, I.1.2 Algorithms

Keywords and phrases population protocol, uniform bipartition, distributed protocol

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.19

1 Introduction

1.1 The Background

A population protocol model [4] is an abstract model that represents computation on a network of low-performance devices. We refer to such devices as agents and a set of agents as a population. Agents can update their states by interacting with other agents, and proceed with computation by repeating the pairwise interactions. The population protocol model can be applied to many systems such as sensor networks and molecular robot networks. For example, one may construct sensor networks to monitor wild birds by attaching sensors to them. In this system, sensors collect and process data based on pairwise interactions when two sensors (or birds) come sufficiently close to each other. Another example is a system of low-performance molecular robots [22]. In this system, a large number of molecular robots compose a network inside a human body and discriminate the physical condition. To realize such systems, many protocols have been proposed as building blocks in the population protocol model [10]. For example, they include leader election protocols [1, 2, 8, 15, 17, 19, 23, 24, 25], counting protocols [9, 11, 12, 20], and majority protocols [1, 3, 6, 18].

In this paper, we consider a uniform bipartition problem, which divides a population into two groups of the same size. The uniform bipartition problem is a special case of a group composition problem, which divides a population into multiple groups to satisfy some conditions. Some protocols for the group composition problem are developed as subroutines to realize fault-tolerant protocols [16] and periodic functions [21]. However, the complexity of the problem has not been studied deeply yet. For this reason, as the first step to study the complexity of the group composition problem, we focus on the space complexity of the uniform bipartition problem. Note that the uniform bipartition problem itself has some applications. For example, we can reduce energy consumption by switching on one group and switching off the other. In another example, we can assign a different task to each group and make agents execute multiple tasks at the same time. This can be regarded as differentiation of a population in the sense that initially identical agents are eventually divided into two groups and execute different tasks. In addition, by repeating uniform bipartition, we can divide a population into an arbitrary number of groups with almost the same size. For example, by repeating uniform bipartition four times, we can make sixteen groups of the same size. We can regroup the sixteen groups to three groups with almost the same size by partitioning them into five, five, and six groups.

1.2 Our Contributions

For the uniform bipartition problem, we clarify solvability and minimum requirement of agent space under various assumptions. More concretely, we consider four types of assumptions, 1) a population with or without a base station, 2) weak or global fairness, 3) symmetric or asymmetric protocols, and 4) designated or arbitrary initial states. A base station (BS) is a distinguishable agent with a powerful capability, and it is useful to realize good properties while it limits the range of application. Fairness is an assumption on interaction patterns of agents. While weak fairness assumes only that interaction occurs infinitely often between each pair of agents, global fairness makes a stronger assumption on the order of interactions (the definition is given in Section 2). Symmetric property of protocols is related to the power of symmetry breaking in the population. Asymmetric protocols may include transitions that make agents with the same states transit to different states. This requires a mechanism to break symmetry among agents and its implementation is sometimes difficult with low-performance agents such as molecular robots. Symmetric protocols do not include such transitions. The assumption of initial states is related to the requirement of initialization and the fault-tolerant property. If a protocol requires a designated initial state, we need to initialize all agents to execute protocols. On the other hand, when the protocol allows arbitrary initial states, initialization of agents other than the BS is not necessary. In addition, even if agents enter arbitrary states due to transient faults, the system can eventually reach the desired configuration by initializing the BS. If a protocol allows arbitrary initial states and does not require a BS, the protocol is self-stabilizing because it can work from arbitrary initial configurations.

For each combination of assumptions, we completely clarify constant-space solvability of the uniform bipartition problem and, if solvable, give a space-optimal protocol (except for protocols given in [16, 14]). The results are shown in Table 1. Each element of the table represents the minimum number of agent states (except for a BS) to solve the uniform bipartition problem on the setting. First, we consider protocols in the case of a single BS. If protocols assume designated initial states, we prove three states are necessary and sufficient. If protocols allow arbitrary initial states, four states are necessary and sufficient under global fairness, while no constant-space protocol exists under weak fairness. Next, we consider

Table 1 The minimum number of states to solve the uniform bipartition problem, where n is the number of agents.

BS	Fairness	Designated initial states		Arbitrary initial states	
		Asymmetric	Symmetric	Asymmetric	Symmetric
Single BS	Globally fair	3	3	4	4
	Weakly fair	3	3	$\Omega(n)$	$\Omega(n)$
No BS	Globally fair	3*	4**	Impossible	Impossible
	Weakly fair	3*	Impossible	Impossible	Impossible

* A protocol with three states is proposed in [16].

** A protocol with four states is obtained by a general transformer in [14].

protocols in the case of no BS. If protocols assume designated initial states, three states are necessary and sufficient for asymmetric protocols. However, if we focus on symmetric protocols, no protocol exists under weak fairness and four states are necessary and sufficient under global fairness. For the case of arbitrary initial states, we prove no protocol exists if we assume no BS. This implies that a BS is necessary for protocols with arbitrary initial states.

1.3 Related Works

The population protocol model was introduced by Angluin et al. [4, 7]. They regard initial states of agents as an input to the system, and resultant states of them as an output from the system. Following this definition, they clarified the class of computable predicates in the population protocol model.

In addition to such computability researches, many algorithmic problems have been considered in the population protocol model. For example, they include leader election [1, 2, 8, 15, 17, 19, 23, 24, 25], counting [9, 11, 12, 20], and majority [1, 3, 6, 18]. These problems are considered under various assumptions of a population with or without a base station, global or weak fairness, symmetric or asymmetric protocols, designated or arbitrary initial states. The leader election problem has been thoroughly studied for both designated and arbitrary initial states. For designated initial states, many researches aim to minimize the time and space complexity [1, 2, 17]. For arbitrary initial states, many papers have developed self-stabilizing and loosely-stabilizing protocols [8, 15, 19, 23, 24, 25]. Cai et al. [15] proposed a self-stabilizing leader election protocol with knowledge of n, and proved that knowledge of n is necessary to construct a self-stabilizing leader election protocol, where n is the number of agents. To overcome the requirement of knowledge of n, Sudo et al. [24] proposed a concept of loose stabilization and gave a loosely-stabilizing leader election protocol. The complexity and the requirement on communication graphs are improved later [19, 23, 25]. The counting problem aims to count the number of agents and it has been studied under assumptions of a single BS and arbitrary initial states. After the first protocol was proposed in [12], the space complexity was gradually minimized [11, 20]. In [9], a time and space optimal protocol was proposed. The majority problem is also a fundamental problem in the population protocol model. In this problem, each agent initially has a color x or y, and the goal is to decide which color gets a majority. For the majority problem, many protocols have been proposed [1, 3, 6, 18]. Recently an asymptotically space-optimal protocol for c colors ($c > 2$) has been proposed in [18].

As a similar problem to the uniform bipartition problem, a group composition problem is studied in [16, 21]. Delporte-Gallet et al. [16] proposed a protocol to divide a population into g groups of the same size. The protocol is asymmetric, assumes designated initial states,

and works under global fairness in the model of no BS. When $g = 2$, the protocol solves the uniform bipartition problem with three states. However, the paper does not consider other setting. Lamani et al. [21] studied a problem that divides a population into groups of designated sizes. Although the proposed protocols assume arbitrary initial states, they also assume that $n/2$ pairs of agents make interactions at the same time and that agents know n. In addition, the protocol requires n states, that is, it is not a constant-space protocol.

2 Definitions

2.1 Population Protocol Model

A population A is defined as a collection of pairwise interacting agents. A protocol is defined as $P = (Q, \delta)$, where Q is a set of possible states of agents and δ is a set of transitions on Q. Each transition in δ is described in the form $(p, q) \to (p', q')$, which means that, when an agent in state p and an agent in state q interact, they change their states to p' and q', respectively. In this paper, only deterministic protocols are considered. If transition $(p, q) \to (p', q')$ satisfies $p = q$ and $p' \neq q'$, the transition is asymmetric; otherwise, the transition is symmetric. For protocol $P = (Q, \delta)$, P is symmetric if every transition in δ is symmetric, and P is asymmetric if every transition in δ is symmetric or asymmetric. Note that a symmetric protocol is also asymmetric.

A global state of a population is called a configuration. A configuration is defined as a vector of (local) states of all agents. We define $s(a, C)$ as the state of agent a at configuration C. When C is clear from the context, we simply write $s(a)$. If configuration C' is obtained from configuration C by a single transition of a pair of agents, we say $C \to C'$. For configurations C and C', if there is a sequence of configurations $C = C_0, C_1, \cdots, C_k = C'$ that satisfies $C_i \to C_{i+1}$ for any i $(0 \leq i < k)$, we say C' is reachable from C, denoted by $C \xrightarrow{*} C'$.

If an infinite sequence of configurations $E = C_0, C_1, C_2, \ldots$ satisfies $C_i \to C_{i+1}$ for any i $(i \geq 0)$, E is an execution of a protocol. An execution E is weakly fair if every pair of agents in A interacts infinitely often. An execution E is globally fair if, for every pair of configurations C and C' such that $C \to C'$, C' occurs infinitely often when C occurs infinitely often. Intuitively, global fairness guarantees that, if configuration C occurs infinitely often, every possible interaction at C occurs infinitely often. If C occurs infinitely often, C' satisfying $C \to C'$ occurs infinitely often, and consequently C'' satisfying $C' \to C''$ also occurs infinitely often. This implies that, under global fairness, if C occurs infinitely often, every configuration C^* reachable from C also occurs infinitely often.

In this paper, we consider two models, one with a single BS (base station) and one with no BS. In the model with a single BS, we assume that a single agent called a BS exists in A. The BS is distinguishable from other non-BS agents while non-BS agents are identical and cannot be distinguished. That is, state set Q is divided into state set Q_b of a BS and state set Q_p of non-BS agents. The BS can be as powerful as needed, in contrast with resource-limited non-BS agents. That is, we focus on the number of states $|Q_p|$ for non-BS agents and do not care the number of states $|Q_b|$ for the BS. In addition, even if we consider protocols with arbitrary initial states, we assume that the BS has a designated initial state while all non-BS agents have arbitrary initial states. If we consider protocols with designated initial states, all non-BS agents have the same designated initial states and the BS has another designated initial state. In the model with no BS, no BS exits and all agents are identical. In this case, they all have the same designated initial states or arbitrary initial states. In both models, no agent knows the total number of agents in the initial configuration.

2.2 Uniform Bipartition Problem

Let A_p be a set of all non-BS agents. Let $f : Q_p \to \{red, blue\}$ be a function that maps a state of a non-BS agent to *red* or *blue*. We define a color of $a \in A_p$ as $f(s(a))$. We say agent $a \in A_p$ is *red* if $f(s(a)) = red$ and agent $a \in A_p$ is *blue* if $f(s(a)) = blue$.

Configuration C is stable if there is a partition $\{R, B\}$ of A_p that satisfies the following condition: 1) $||R| - |B|| \leq 1$, and 2) for every C^* such that $C \xrightarrow{*} C^*$, each agent in R is *red* and each agent in B is *blue* at C^*.

An execution $E = C_0, C_1, C_2, \ldots$ solves the uniform bipartition problem if there is a stable configuration C_t in E. If each execution E of protocol P solves the uniform bipartition problem, we say protocol P solves the uniform bipartition problem. The main objective of this paper is to minimize the number of states for non-BS agents. Since the BS is powerful, we do not care the number of states for the BS. When protocol P requires x states for non-BS agents, we say P is a protocol with x states.

For simplicity, we use agents only to refer to non-BS agents in the following sections. To refer to the BS, we always use the BS (not an agent).

3 Uniform Bipartition Protocols with a Single BS

In this section, we consider the uniform bipartition problem under the assumption of a single BS. Recall that the BS is distinguishable from other non-BS agents, and we do not care the number of states for the BS.

3.1 Protocols with Designated Initial States

In this subsection, we consider protocols with designated initial states. We give a simple symmetric protocol with three states under global or weak fairness, and then prove that there exists no asymmetric protocol with two states under global or weak fairness. This implies that, in this case, three states are sufficient for asymmetric or symmetric protocols under global or weak fairness.

3.1.1 A protocol with three states

In this protocol, the state set of (non-BS) agents is $Q_p = \{initial, red, blue\}$, and we set $f(initial) = f(red) = red$ and $f(blue) = blue$. The designated initial state of all agents is *initial*. The idea of the protocol is to assign states *red* and *blue* to agents alternately when agents interact with the BS. To realize this, the BS has a state set $Q_b = \{b_{red}, b_{blue}\}$, and its initial state is b_{red}. The protocol consists of the following two transitions.

1. $(b_{red}, initial) \to (b_{blue}, red)$
2. $(b_{blue}, initial) \to (b_{red}, blue)$

That is, when the BS in state b_{red} (resp., b_{blue}) and a non-BS agent in state *initial* interact, the BS changes the state of the non-BS agent to *red* (resp., *blue*) and the state of itself to b_{blue} (resp., b_{red}). When two non-BS agents interact, no state transition occurs. Clearly, all non-BS agents evenly transit to state *red* or *blue*, and the difference in the numbers of *red* and *blue* agents is at most one. Note that the protocol contains no asymmetric transition and works correctly if every non-BS agent interacts with the BS. Therefore, we have the following theorem.

▶ **Theorem 1.** *In the model with a single BS, there exists a symmetric protocol with three states and designated initial states that solves the uniform bipartition problem under global or weak fairness.*

3.1.2 Impossibility with two states

Next, we show three states are necessary to construct an asymmetric protocol under global or weak fairness. This implies that, in this case, three states are necessary for asymmetric or symmetric protocols under global or weak fairness because a symmetric protocol is also asymmetric. That is, three states are necessary and sufficient in this case.

▶ **Theorem 2.** *In the model with a single BS, no asymmetric protocol with two states and designated initial states solves the uniform bipartition problem under global or weak fairness.*

Proof. We prove that such a protocol does not exist even if its execution satisfies both global and weak fairness. For contradiction, assume that such a protocol Alg exists. Without loss of generality, we assume $Q_p = \{s_1, s_2\}, f(s_1) = red, f(s_2) = blue$, and that the designated initial state of all agents is s_1. Let n is an even number that is at least four. We consider the following three cases.

First, for population A of a single BS and n (non-BS) agents $a_1, a_2, \ldots, a_n$, consider an execution $E = C_0, C_1, \ldots$ of Alg that satisfies both global and weak fairness. According to the definition, there exists a stable configuration C_t. That is, after C_t, the state of each agent does not change even if the BS and agents in states s_1 and s_2 interact in any order.

Next, for population A' of a single BS and $n+2$ agents $a_1, a_2, \ldots, a_{n+2}$, we define an execution $E' = C'_0, C'_1, \ldots, C'_t, C'_{t+1}, \ldots$ of Alg as follows.

- From C'_0 to C'_t, the BS and n agents $a_1, a_2, \ldots, a_n$ interact in the same order as the execution E.
- After C'_t, the BS and $n+2$ agents interact so as to satisfy both global and weak fairness.

Since the BS and agents $a_1, \ldots, a_n$ change their states similarly to E from C'_0 to C'_t, there are $n/2+2$ agents in state s_1 and $n/2$ agents in state s_2 at C'_t. Moreover, the state of the BS at C'_t is the same as the state of the BS at C_t. However, since the difference in the numbers of *red* and *blue* agents is two, C'_t is not a stable configuration. Consequently, after C'_t, some *red* or *blue* agent changes its state in execution E'.

Lastly, we consider execution E for population A again. Here, we consider interactions after stable configuration C_t, and apply interactions in E' to execution E. That is, we consider the following execution after C_t: 1) when the BS and an agent in state $s \in \{s_1, s_2\}$ interact at $C'_u \to C'_{u+1}$ $(u \geq t)$ in E', the BS and an agent in state s interact at $C_u \to C_{u+1}$ in E, and 2) when two agents in states $s \in \{s_1, s_2\}$ and $s' \in \{s_1, s_2\}$ interact at $C'_u \to C'_{u+1}$ $(u \geq t)$ in E', two agents in states s and s' interact at $C_u \to C_{u+1}$ in E. We can construct such an execution because, after stable configuration C_t, at least two agents are in s_1 and at least two agents are in s_2. In this execution E, since interactions occur similarly to E', some *red* or *blue* agent changes its state similarly to E' after C_t. This is a contradiction because C_t is a stable configuration. ◀

3.2 Protocols with Arbitrary Initial States

In this subsection, we consider protocols with arbitrary initial states. Recall that, since a BS is powerful, the BS can start the protocol from a designated initial state.

3.2.1 Under global fairness

Under global fairness, we give a symmetric protocol with four states, and prove impossibility of protocols with three states. That is, we show that four states are necessary and sufficient to construct a (symmetric or asymmetric) protocol in this case.

3.2.1.1 A symmetric protocol with four states

Here we show a symmetric protocol with four states under global fairness. In this protocol, each (non-BS) agent x has two variables $rb_x \in \{red, blue\}$ and $mark_x \in \{0, 1\}$. Variable rb_x represents the color of agent x. That is, for state s of agent x, $f(s) = red$ holds if $rb_x = red$ and $f(s) = blue$ holds if $rb_x = blue$. We define $\#red$ as the number of red agents and $\#blue$ as $blue$ agents. We explain the role of variable $mark_x$ later.

The basic strategy of the protocol is that the BS counts red and $blue$ agents by counting protocol $Count$ [11] and changes colors of agents so that the numbers of red and $blue$ agents become equal. Protocol $Count$ is a symmetric protocol that counts the number of non-BS agents from arbitrary initial states under global fairness. Protocol $Count$ uses only two states for each non-BS agent. In protocol $Count$, the BS has variable $Count.out$ that eventually outputs the number of agents. More concretely, $Count.out$ initially has value 0, gradually increases one by one, eventually equals to the number of agents, and stabilizes. The following lemma explains the characteristic of protocol $Count$.

▶ **Lemma 3** ([11]). *Let n be the number of non-BS agents. In the initial configuration, $Count.out = 0$ holds. When $Count.out < n$, $Count.out$ eventually increases by one under global fairness. When $Count.out = n$, $Count.out$ never changes and stabilizes.*

To count red and $blue$ agents, the BS executes two instances of protocol $Count$ in parallel to the main procedure of the uniform bipartition protocol. We denote by $Count_{red}$ and $Count_{blue}$ instances of protocol $Count$ to count red and $blue$ agents, respectively. The BS executes $Count_{red}$ when it interacts with a red agent. That is, the BS updates variables of $Count_{red}$ at the BS and the red agent by applying a transition of protocol $Count_{red}$. By this behavior, the BS executes $Count_{red}$ as if the population contains only red agents. Therefore, after the BS initializes its own variables of $Count_{red}$, it can correctly count the number of red agents by $Count_{red}$ (i.e., $Count_{red}.out$ eventually stabilizes to $\#red$) as long as a set of red agents does not change. Similarly, the BS executes $Count_{blue}$ when it interacts with a $blue$ agent, and counts the number of $blue$ agents. The straightforward approach to use the counting protocols is to adjust colors of agents after $Count_{red}.out$ and $Count_{blue}.out$ stabilize. However, the BS cannot know whether the outputs have stabilized or not. For this reason, the BS maintains estimated numbers of red and $blue$ agents, and it changes colors of agents when the difference in the estimated numbers of red and $blue$ agents is two. Note that, since the counting protocols assume that a set of counted agents does not change, the BS must restart $Count_{red}$ and $Count_{blue}$ from the beginning when the BS changes colors of some agents.

We explain the details of this procedure. The BS records the estimated numbers of red and $blue$ agents in variables $C^*_{rb}[red]$ and $C^*_{rb}[blue]$, respectively. In the beginning of execution, these variables are identical to outputs of $Count_{red}$ and $Count_{blue}$. If the difference between $C^*_{rb}[red]$ and $C^*_{rb}[blue]$ becomes two, the BS immediately changes colors of agents. At the same time, the BS updates $C^*_{rb}[red]$ and $C^*_{rb}[blue]$ to reflect the change of colors. After the BS changes colors of some agents, it restarts $Count_{red}$ and $Count_{blue}$ from the beginning by initializing its own variables of the counting protocols. Since the counting protocols allow

Algorithm 1 Uniform bipartition protocol.

Variables at BS:
$C^*_{rb}[c](c \in \{red, blue\})$: the estimated number of c agents, initialized to 0
$Variables$: variables of $Count_c(c \in \{red, blue\})$
Variables at a mobile agent x:
$rb_x \in \{red, blue\}$: color of the agent, initialized arbitrarily
$mark_x \in \{0, 1\}$: a variable of $Count_c(c \in \{red, blue\})$, initialized arbitrarily
1: **when** a mobile agent x interacts with BS **do**
2: update $mark_x$ and variables of $Count_{rb_x}$ at BS by applying a transition of $Count_{rb_x}$
3: **if** $C^*_{rb}[rb_x] < Count_{rb_x}.out$ **then**
4: $C^*_{rb}[rb_x] \leftarrow Count_{rb_x}.out$
5: **end if**
6: **if** $C^*_{rb}[rb_x] - C^*_{rb}[\overline{rb_x}] > 1$ **then**
7: $C^*_{rb}[rb_x] \leftarrow C^*_{rb}[rb_x] - 1$
8: $C^*_{rb}[\overline{rb_x}] \leftarrow C^*_{rb}[\overline{rb_x}] + 1$, $rb_x \leftarrow \overline{rb_x}$
9: reset variables of $Count_{red}$ and $Count_{blue}$ at BS
10: **end if**
11: **end when**

arbitrary initial states of non-BS agents, the BS can correctly count *red* and *blue* agents after that. Note that the BS does not initialize $C^*_{rb}[red]$ and $C^*_{rb}[blue]$ because it knows such numbers of *red* and *blue* agents exist. If the output of $Count_{red}$ and $Count_{blue}$ exceeds $C^*_{rb}[red]$ and $C^*_{rb}[blue]$, the BS updates $C^*_{rb}[red]$ and $C^*_{rb}[blue]$, respectively. After that, if the difference between $C^*_{rb}[red]$ and $C^*_{rb}[blue]$ becomes two, the BS changes colors of agents. By repeating this behavior, the BS adjusts colors of agents.

The pseudocode of this protocol is given in Algorithm 1. We define $\overline{red} = blue$ and $\overline{blue} = red$. Variable $mark_x$ is a two-state variable of counting protocols $Count_{red}$ and $Count_{blue}$. Since the BS restarts the counting protocols whenever it changes colors of agents, the BS keeps a set of *red* (resp., *blue*) agents unchanged until it restarts $Count_{red}$ (resp., $Count_{blue}$). In addition, each agent is involved in either $Count_{red}$ or $Count_{blue}$ at the same time. Hence it requires only a single variable $mark_x$ to execute $Count_{red}$ and $Count_{blue}$. When two non-BS agents interact, no state transition occurs in this protocol and counting protocols. When the BS and a *red* agent interact, they update $mark_x$ and variables of $Count_{red}$ at the BS by applying a transition of $Count_{red}$. This means that they execute $Count_{red}$ in parallel to the main procedure of the uniform bipartition protocol. After that, if $Count_{red}.out$ is larger than $C^*_{rb}[red]$, $C^*_{rb}[red]$ is updated with $Count_{red}.out$. If $C^*_{rb}[red]$ is larger than $C^*_{rb}[blue]$ by two or more, the *red* agent changes its color to *blue* and the BS updates $C^*_{rb}[red]$ and $C^*_{rb}[blue]$. After updating, the BS resets variables of $Count_{red}$ and $Count_{blue}$, and restarts counting. When the BS and a *blue* agent interact, they behave similarly.

In the following, we prove the correctness of Algorithm 1.

▶ **Lemma 4.** *In any configuration, $C^*_{rb}[red] \leq \#red$, $C^*_{rb}[blue] \leq \#blue$ and $|C^*_{rb}[red] - C^*_{rb}[blue]| \leq 1$ hold.*

Proof. We prove by induction on the index $k \geq 0$ of a configuration in an execution $C_0, C_1, C_2, \cdots, C_k, \cdots$. At the initial configuration C_0, the lemma holds. Let us assume that the lemma holds for configuration C_k and prove it for configuration C_{k+1}. From this assumption, $C^*_{rb}[red] \leq \#red$, $C^*_{rb}[blue] \leq \#blue$ and $|C^*_{rb}[red] - C^*_{rb}[blue]| \leq 1$ hold at C_k.

Assume that, when C_k transits to C_{k+1}, the BS and agent x interact. If $Count_{rb_x}.out$ becomes larger than $C^*_{rb}[rb_x]$, the BS updates $C^*_{rb}[rb_x]$ by $C^*_{rb}[rb_x] \leftarrow Count_{rb_x}.out$ (line 3). Note that, in this case, $C^*_{rb}[rb_x]$ increases by one from Lemma 3. In addition, $C^*_{rb}[red] \leq \#red$ and $C^*_{rb}[blue] \leq \#blue$ still hold. Recall that $|C^*_{rb}[red] - C^*_{rb}[blue]| \leq 1$ held before this update and $C^*_{rb}[rb_x]$ increases by one. Conseqently, at this moment (before line 5), $|C^*_{rb}[rb_x] - C^*_{rb}[\overline{rb_x}]| \leq 1$ or $C^*_{rb}[rb_x] - C^*_{rb}[\overline{rb_x}] = 2$ holds. Next, we consider lines 5 to 9. If $C^*_{rb}[rb_x] - C^*_{rb}[\overline{rb_x}] \leq 1$ at line 5, lines 6 to 8 are not executed, and thus $C^*_{rb}[red] \leq \#red$, $C^*_{rb}[blue] \leq \#blue$ and $|C^*_{rb}[red] - C^*_{rb}[blue]| \leq 1$ hold. If $C^*_{rb}[rb_x] - C^*_{rb}[\overline{rb_x}] = 2$ at line 5, agent x changes its color from rb_x to $\overline{rb_x}$, $C^*_{rb}[rb_x]$ decreases by one, and $C^*_{rb}[\overline{rb_x}]$ increases by one. This also preserves $C^*_{rb}[red] \leq \#red$, $C^*_{rb}[blue] \leq \#blue$ and $|C^*_{rb}[red] - C^*_{rb}[blue]| \leq 1$. Therefore, the lemma holds. ◀

▶ **Theorem 5.** *Algorithm 1 solves the uniform bipartition problem. That is, in the model with a BS, there exists a symmetric protocol with four states and arbitrary initial states that solves the uniform bipartition problem under global fairness.*

Proof. We define $phase = C^*_{rb}[red] + C^*_{rb}[blue]$. Initially, $phase = 0$ holds. We show that 1) $phase$ increases one by one if $phase < n$, and 2) Algorithm 1 solves the uniform bipartition problem if $phase = n$.

First consider the initial configuration. Since we assume global fairness, $Count_{red}.out$ or $Count_{blue}.out$ increases by one from Lemma 3 and at that time $phase$ increases by one.

Let us consider the transition $C \to C'$ such that $phase$ increases by one (i.e., line 4 is executed) and $phase < n$ holds at C'. We consider two cases.

- Case that lines 7 to 9 are not executed at $C \to C'$. In this case, since the BS does not change sets of red and $blue$ agents, it can correctly continue to execute $Count_{red}$ and $Count_{blue}$. Since $phase < n = \#red + \#blue$ holds, either $\#red > C^*_{rb}[red]$ or $\#blue > C^*_{rb}[blue]$ holds. Consequently, from Lemma 3, either $Count_{red}.out > C^*_{rb}[red]$ or $Count_{blue}.out > C^*_{rb}[blue]$ holds eventually because we assume global fairness. At that time, $C^*_{rb}[red]$ or $C^*_{rb}[blue]$ increases by one and hence $phase$ increases by one.
- Case that lines 7 to 9 are executed at $C \to C'$. In this case, the BS changes sets of red and $blue$ agents. At that time, the BS initializes its own variables of counting algorithms $Count_{red}$ and $Count_{blue}$. Since the counting algorithms work from arbitrary initial states of agents, the BS can correctly execute $Count_{red}$ and $Count_{blue}$ from the beginning under global fairness. Similarly to the first case, from Lemma 3, either $Count_{red}.out > C^*_{rb}[red]$ or $Count_{blue}.out > C^*_{rb}[blue]$ holds eventually. Then, $phase$ increases by one.

Lastly, consider the transition $C \to C'$ such that $phase$ increases by one and $phase = n$ holds at C'. From $phase = n$, $C^*_{rb}[red] + C^*_{rb}[blue] = n = \#red + \#blue$ holds, and consequently $C^*_{rb}[red] = \#red$ and $C^*_{rb}[blue] = \#blue$ hold from Lemma 4. This implies that $Count_{red}.out$ and $Count_{blue}.out$ never exceed $C^*_{rb}[red]$ and $C^*_{rb}[blue]$ after that, respectively. Therefore, $C^*_{rb}[red]$ and $C^*_{rb}[blue]$ are never updated and consequently agents never change their colors any more. Since $|\#red - \#blue| = |C^*_{rb}[red] - C^*_{rb}[blue]| \leq 1$ holds from Lemma 4, we have the theorem. ◀

3.2.1.2 Impossibility with three states

Here we show the impossibility of asymmetric protocols with three states.

▶ **Theorem 6.** *In the model with a single BS, no asymmetric protocol with three states and arbitrary initial states solves the uniform bipartition problem under global fairness.*

Proof. For contradiction, assume that such a protocol Alg exists. Without loss of generality, we assume that the state set of agents is $Q_p = \{s_1, s_2, s_3\}$, $f(s_1) = f(s_2) = red$, and $f(s_3) = blue$. We consider the following three cases.

First, consider population $A = \{a_0, \ldots, a_n\}$ of a single BS and n agents such that n is even and at least 4. Assume that a_0 is a BS. Since each agent has an arbitrary initial state, we consider an initial configuration C_0 such that $s(a_i) = s_3$ holds for any $i (1 \leq i \leq n)$. Note that the BS a_0 has a designated initial state at C_0. From the definition of Alg, for any globally fair execution $E = C_0, C_1, \cdots$, there exists a stable configuration C_t. Hence, both the number of *red* agents and the number of *blue* agents are $n/2$ at C_t. After C_t, the color of agent a_i (i.e., $f(s(a_i))$) never changes for any $a_i (1 \leq i \leq n)$ even if the BS and agents interact in any order.

Next, consider population $A' = \{a'_0, \ldots, a'_{n+2}\}$ of a single BS and $n+2$ agents. Assume that agent a'_0 is a BS. We consider an initial configuration C'_0 such that $s(a'_i) = s_3$ holds for any i $(1 \leq i \leq n+2)$. From this initial configuration, we define an execution $E' = C'_0, C'_1, \cdots, C'_t, \cdots$ using the execution E as follows.

- For $0 \leq u < t$, when a_i and a_j interact at $C_u \to C_{u+1}$, a'_i and a'_j interact at $C'_u \to C'_{u+1}$.
- For $t \leq u$, an interaction occurs at $C'_u \to C'_{u+1}$ so that E' satisfies global fairness.

Since the BS and agents $a_1, \ldots, a_n$ change their states similarly to E from C'_0 to C'_t, $s(a'_i) = s(a_i)$ holds for $1 \leq i \leq n$. Hence, there exist $n/2$ *red* agents and $n/2+2$ *blue* agents at C'_t. Consequently C'_t is not a stable configuration. This implies that there exists a stable configuration $C'_{t'}$ for some $t' > t$. Clearly at least one *blue* agent becomes *red* from C'_t to $C'_{t'}$. That is, for some configuration $C'_{t^*} (t \leq t^* < t')$, an agent in state s_3 transits to state s_1 or s_2 at $C_{t^*} \to C_{t^*+1}$. Assume that t^* is the smallest value that satisfies the condition.

Finally, for A we define an execution $E'' = C''_0, C''_1, \cdots$ using executions E and E' as follows.

- Let $C''_u = C_u$ for $0 \leq u \leq t$. That is, E'' reaches stable configuration C''_t in similarly to E.
- For $t \leq u \leq t^*$, we define an execution so that interaction at $C'_u \to C'_{u+1}$ also occurs at $C''_u \to C''_{u+1}$. Concretely, when a'_i and a'_j interact at $C'_u \to C'_{u+1}$, we define $a_{i'}$ and $a_{j'}$ as follows and they interact at $C''_u \to C''_{u+1}$. If $i \leq n$, let $i' = i$. Otherwise, since $s(a'_i) = s_3$ holds at C'_u (because no agent in state s_3 changes its state from C'_t to C'_{t^*}), choose $i' (\leq n)$ such that both $s(a_{i'}) = s_3$ and $i' \neq j$ hold. Similarly, if $j \leq n$, let $j' = j$. Otherwise choose $j' (\leq n)$ such that both $s(a_{j'}) = s_3$ and $j' \neq i'$ hold. Such i' and j' exist since at least two agents in state s_3 exist (because $n \geq 4$ holds and no agent in state s_3 changes its state from C'_t to C'_{t^*}).

Clearly, for $t \leq u \leq t^*$ and $i \leq n$, $s(a_i)$ at C''_u is equal to $s(a'_i)$ at C'_u. Additionally, at $C''_{t^*} \to C''_{t^*+1}$, an agent in state s_3 transits to s_1 or s_2 as well as $C'_{t^*} \to C'_{t^*+1}$. This means that the agent changes its color at $C''_{t^*} \to C''_{t^*+1}$, which contradicts that C''_t is a stable configuration. ◀

▶ Remark. Recall that Section 3.1.1 gives a protocol with three states and designated initial states. In the protocol, the state set of agents is $Q_p = \{initial, red, blue\}$, we set $f(initial) = f(red) = red$ and $f(blue) = blue$, and the designated initial state is $initial$. The important point is that the designated initial state (i.e., $initial$) has the same color as one of other states (i.e., red).

In the proof of Theorem 6, we consider an execution such that all agents have the same initial state in the initial configuration. The difference from the above protocol is that the initial state does not have the same color as any other state. This means, even if we consider

a protocol with three states and designated initial states, there exists no protocol such that the designated initial state does not have the same color as any other state. This fact holds even if the number of states is larger than three. ◀

3.2.2 Under weak fairness

Under weak fairness, we prove that no protocol with constant states solves the uniform bipartition problem. To prove this impossibility, we borrow techniques used in the impossibility proof for the counting problem [12]. This work shows that, in the model with a single BS, when the upper bound of the number of non-BS agents is n, no asymmetric protocol with $n-2$ states and arbitrary initial states solves the counting problem under weak fairness. We can apply the proof in [12] to the uniform bipartition problem in a straightforward manner.

▶ **Theorem 7.** *Let n be an even number that is at least four. In the model with a single BS, when the upper bound of the number of non-BS agents is n, no asymmetric protocol with $n-3$ states and arbitrary initial states solves the uniform bipartition problem under weak fairness.*

Proof. For contradiction, assume that such a protocol *Alg* exists. We consider the following two cases.

First, consider population $A = \{a_0, \ldots, a_{n-2}\}$ of a single BS and $n-2$ agents such that a_0 is a BS. We consider an initial configuration C_0 such that initial states of $a_0, \ldots, a_{n-2}$ are $s_0, \ldots, s_{n-2}$ (s_0 is a designated initial state of the BS). Since the upper bound of the number of non-BS agents is n and agents do not know the number of agents, *Alg* should work correctly even if the number of non-BS agents is $n-2$. This implies that, for any execution $E = C_0, C_1, \cdots, C_t, \cdots$, there exists a stable configuration C_t. Since the number of states for non-BS agents is $n-3$, there exists y, a_p, and $a_{p'}$ such that configurations satisfying $y = s(a_p) = s(a_{p'})$ appear infinitely many times after C_t.

Next, consider population $A' = \{a'_0, \ldots, a'_n\}$ of a single BS and n agents such that a'_0 is a BS. We consider an initial configuration C_0 such that initial states of $a'_0, \ldots, a'_n$ are $s_0, \ldots, s_{n-2}, y, y$, respectively. For A' we define an execution $E' = C'_0, C'_1, \cdots, C'_t, \cdots$ using the execution E as follow.

- For $0 \leq u \leq t-1$, when a_i and a_j interact at $C_u \to C_{u+1}$, a'_i and a'_j interact at $C'_u \to C'_{u+1}$.

Clearly, $s(a'_i) = s(a_i)$ holds at C'_t for any i ($0 \leq i \leq n-2$). Since $s(a'_n) = s(a'_{n-1}) = y$ holds at C'_t, the difference in the numbers of *red* and *blue* agents remains two and consequently C'_t is not a stable configuration.

After C'_t, we define an execution as follows. This definition aims to make $n-2$ agents behave similarly to E and two agents keep state y.

- Until $y = s(a'_p) = s(a'_{p'})$ holds, if a_i and a_j interact at $C_u \to C_{u+1}$, a'_i and a'_j interact at $C'_u \to C'_{u+1}$.
- To define the remainder of E', we first define procedure $Proc(q, q')$, which creates a sub-execution from two indices q and q'. Procedure $Proc(q, q')$ can be applied to a configuration such that $y = s(a'_p) = s(a'_{p'}) = s(a'_{n-1}) = s(a'_n)$ holds. After that, $Proc(q, q')$ creates a sub-execution similar to E such that all agents in $A(q, q') = (A' - \{a'_p, a'_{p'}, a'_{n-1}, a'_n\}) \cup \{a'_q, a'_{q'}\}$ interact each other and the last configuration also satisfies the above condition. The concrete definition of $Proc(q, q')$ is as follows. When a_i and a_j interact at $C_u \to C_{u+1}$, a'_i and a'_j interact at $C'_u \to C'_{u+1}$ if $i, j \notin \{p, p'\}$. If $i = p$ or $j = p$, a'_q joins the interaction instead of a'_p. If $i = p'$ or $j = p'$, $a'_{q'}$ joins the interaction

instead of $a'_{p'}$. Procedure $Proc(q, q')$ continues these behaviors until all agents in $A(q, q')$ interact each other and satisfy $s(a'_q) = s(a'_{q'}) = y$.

By using $Proc(q, q')$, we define the remainder of E' to satisfy weak fairness as follows: Repeat $Proc(p, p')$, $Proc(p, n-1)$, $Proc(p, n)$, $Proc(n-1, p')$, $Proc(n, p')$, and $Proc(n, n-1)$.

Clearly, E' makes $n-2$ agents behave similarly to E and two agents keep state y. Hence, E' never converges to a stable configuration. Since E' is weakly fair, this is a contradiction. ◀

▶ Remark. Theorem 7 implies that no protocol with at most $n-4$ states solves the uniform bipartition problem under the same assumption. This is because, if a protocol with n_s states ($n_s \leq n-4$) is given, we can transform it to a protocol with $n-3$ states by adding $n-3-n_s$ dummy states. Hence, at least $n-2$ states are necessary to solve the uniform bipartition problem under this assumption.

On the other hand, the sufficient number of states to solve the uniform bipartition problem under this assumption is not known. To clarify the matching lower and upper bounds of the number of states is an open problem. ◀

4 Uniform Bipartition Protocols with No BS

In this section, we consider the uniform bipartition problem under the assumption of no BS. That is, all agents are identical.

4.1 Protocols with Designated Initial States

In this subsection, we consider protocols with designated initial states. Since we consider the model with no BS, all agents have the same initial state in the initial configuration.

4.1.1 Asymmetric protocols

First, we consider asymmetric protocols in this case. Since three states are necessary in the model with a BS from Theorem 2, three states are also necessary in the model with no BS. In addition, Delporte-Gallet et al. [16] gives a protocol with three states. This implies that three states are necessary and sufficient in this case.

Here, we briefly explain the protocol proposed in [16]. In this protocol, the state set of agents is $Q_p = \{initial, red, blue\}$, and we set $f(initial) = f(red) = red$ and $f(blue) = blue$. The designated initial state of all agents is $initial$. The protocol consists of a single asymmetric transition $(initial, initial) \to (red, blue)$. In this protocol, when two agents in state $initial$ interact, one agent transits to red and the other transits to $blue$. This implies that the number of agents in state red is always the same as the number of agents in state $blue$. Eventually all agents (possibly except one agent) transit to state red or $blue$. From $f(initial) = red$, the difference in the numbers of red and $blue$ agents is at most one. Note that the protocol works correctly if every pair of agents interacts once.

▶ **Theorem 8** ([16]). *In the model with no BS, there exists an asymmetric protocol with three states and designated initial states that solves the uniform bipartition problem under global or weak fairness.*

4.1.2 Symmetric protocols

Next, we consider symmetric protocols in this case. For this setting, we give three results: 1) a protocol with four states under global fairness, 2) impossibility with three states under global fairness, and 3) impossibility under weak fairness. These results show that, in this case, four states are necessary and sufficient to construct a symmetric protocol under global fairness, and no symmetric protocol exists under weak fairness.

4.1.2.1 A protocol with four states under global fairness

We can easily obtain a symmetric protocol with four states by a scheme proposed in [14]. The scheme transforms an asymmetric protocol with α states to a symmetric protocol with at most 2α states. By applying the scheme to an asymmetric protocol in Section 4.1.1 and deleting unnecessary states, we can obtain a symmetric protocol with four states.

For self-containment, we briefly explain the obtained protocol. Since no symmetric protocol solves the uniform bipartition problem for a population of two agents, we assume that a population consists of at least three agents. In this protocol, the state set of agents is $Q_p = \{initial, initial', red, blue\}$, and we set $f(initial) = f(initial') = f(red) = red$ and $f(blue) = blue$. The designated initial state of all agents is $initial$. The protocol consists of the following seven transitions.

1. $(initial, initial) \to (initial', initial')$
2. $(initial', initial') \to (initial, initial)$
3. $(initial, initial') \to (red, blue)$
4. $(initial, red) \to (initial', red)$
5. $(initial, blue) \to (initial', blue)$
6. $(initial', red) \to (initial, red)$
7. $(initial', blue) \to (initial, blue)$

The main behavior of the protocol is similar to the previous asymmetric protocol with three states. However, since asymmetric transition $(initial, initial) \to (red, blue)$ is not allowed in symmetric protocols, the scheme in [14] introduces a new state $initial'$. Transition 3 implies that, when agents in states $initial$ and $initial'$ interact, they become red and $blue$, respectively. In addition, agents in states $initial$ and $initial'$ become $initial'$ and $initial$ respectively when they interact with some agents (except for interaction between one in state $initial$ and one in state $initial'$). From global fairness, if at least two agents are in state $initial$ or $initial'$, some two agents eventually enter states $initial$ and $initial'$. After that, if the two agents interact, they enter states red and $blue$.

Figure 1 shows an example execution of the protocol for a population of four agents. Initially all agents are in state $initial$ (Fig. 1 (a)). After interactions (a_1, a_2) and (a_3, a_4), all agents enter state $initial'$ (Fig. 1 (b)). Similarly, after interactions (a_1, a_4), (a_2, a_3), (a_1, a_3), and (a_2, a_4), all agents have the same state (Fig. 1 (c) and (d)). If these interactions happen infinite times, all agents keep the same state and never achieve the uniform bipartition. However, under the global fairness, such interactions do not happen infinite times. This is because, if some configuration C occurs infinite times, every configuration reachable from C should occur. This implies that, before a configuration in Fig. 1 (d) occurs infinite times, interactions (a_1, a_2) and (a_1, a_3) happen in this order from the configuration. Then, a_1 and a_3 enter states red and $blue$, respectively (Fig. 1 (e) and (f)). After that, in a similar way, the remaining agents eventually enter red and $blue$ like Fig. 1 (g) and (h).

Theorem 8 and correctness of the scheme in [14] derives the following theorem.

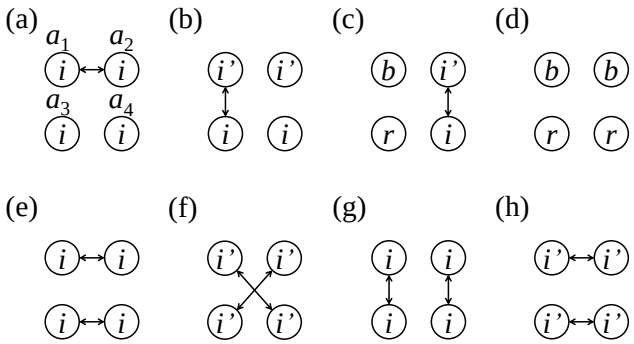


Figure 1 An example execution of the protocol. Symbols i, i', r, and b represent states *initial*, *initial*$'$, *red*, and *blue*, respectively. Arrows represent interactions of agents.

▶ **Theorem 9.** *In the model with no BS, when the number of agents is at least three, there exists a symmetric protocol with four states and designated initial states that solves the uniform bipartition problem under global fairness.*

4.1.2.2 Impossibility results

In the following, we show two impossibility results.

▶ **Theorem 10.** *In the model with no BS, no symmetric protocol with three states and designated initial states solves the uniform bipartition problem under global fairness.*

Proof. For contradiction, assume that such a protocol *Alg* exists. Without loss of generality, we assume that the state set of agents is $Q_p = \{s_1, s_2, s_3\}$, $f(s_1) = f(s_2) = red$, and $f(s_3) = blue$. Consider population $A = \{a_1, \ldots, a_n\}$ of n agents such that n is even and at least 6. First, assume that the designated initial state of all agents is s_3. Clearly, *Alg* has transition $(s_3, s_3) \to (s_i, s_i)$ for some $i \neq 3$. However, since $n/2$ agents in state s_3 exist at a stable configuration, some agents change their states from s_3 to s_i at the stable configuration. This implies that agents change their colors. Therefore, a designated initial state is s_1 or s_2.

Next, assume that the designated initial state of all agents is s_1 (Case of s_2 is the same). Since *Alg* is a symmetric protocol and all the initial states are s_1, *Alg* includes $(s_1, s_1) \to (s_i, s_i)$ for some $i \neq 1$. This implies that all agents can transit to state s_i from the initial configuration. Hence, *Alg* also includes $(s_i, s_i) \to (s_j, s_j)$ for some $j \neq i$. When $i = 3$, since $n/2$ *blue* agents exist at a stable configuration and they are in state s_3, the *blue* agents become *red* by transition $(s_3, s_3) \to (s_j, s_j)$. Therefore, $i \neq 3$ holds.

The remaining case is $i = 2$. If $j = 3$, that is, *Alg* includes $(s_2, s_2) \to (s_3, s_3)$, *red* agents (i.e., agents in state s_1 or s_2) change their colors at a stable configuration because *Alg* includes $(s_1, s_1) \to (s_2, s_2)$ and $(s_2, s_2) \to (s_3, s_3)$. This implies $j = 1$. In this case, *Alg* includes $(s_2, s_2) \to (s_1, s_1)$. Since some agents should transit to state s_3, *Alg* includes $(s_1, s_2) \to (s_k, s_l)$ such that k or l is 3. At a stable configuration, there exist $n/2$ agents with states s_1 or s_2. However, these agents can transit to state s_3 from transitions $(s_1, s_2) \to (s_k, s_l)$, $(s_2, s_2) \to (s_1, s_1)$, and $(s_1, s_1) \to (s_2, s_2)$. This is a contradiction. ◀

▶ **Theorem 11.** *In the model with no BS, no symmetric protocol with designated initial states solves the uniform bipartition problem under weak fairness.*

Proof. For contradiction, assume that such a protocol *Alg* exists. We assume that the state set of agents is $Q_p = \{s_1, s_2, \ldots\}$. Consider population $A = \{a_1, \ldots, a_n\}$ of n agents such

that n is even and at least 2. Let s_{i_1} be the designated initial state of all agents, that is, $s(a_i) = s_{i_1}$ holds for any i $(1 \leq i \leq n)$ at the initial configuration. Clearly, symmetric protocol Alg has transition $(s_{i_1}, s_{i_1}) \to (s_{i_2}, s_{i_2})$ for some s_{i_2}. This implies that, if all pairs of two agents in state s_{i_1} interact, all agents transit to s_{i_2}. Similarly, if all pairs of two agents in state s_{i_2} interact, all agents transit to the same state (say s_{i_3}).

When the above execution is repeated, configurations such that all agents have the same state appear infinitely often. By changing pairs of two agents, we can make the above execution under weak fairness. If all agents are in the same state, such a configuration is not stable because the colors of all agents are the same. This is a contradiction. ◀

4.2 Protocols with Arbitrary Initial States

In this subsection, we consider protocols with arbitrary initial states. We show that, in this case, no protocol solves the uniform bipartition problem. That is, to allow agents to start from arbitrary initial states, a single BS is necessary.

▶ **Theorem 12.** *In the model with no BS, no asymmetric protocol with arbitrary initial states solves the uniform bipartition problem under global fairness*

Proof. For contradiction, assume that such a protocol Alg exists. Assume that n is even and at least 4. We consider the following two cases.

First, for population $A = \{a_1, \ldots, a_n\}$ of n agents, consider an execution $E = C_0, C_1, \cdots$ of Alg. From the definition of Alg, there exists a stable configuration C_t. Hence, both the number of *red* agents and the number of *blue* agents are $n/2$ at C_t. After C_t, the color of agent a_i (i.e.,$f(s(a_i))$) never changes for any a_i $(1 \leq i \leq n)$ even if agents interact in any order.

Next, for population $A' = \{a'_i | f(s(a_i, C_t)) = red\}$ of $n/2$ agents, consider an execution $E' = C'_0, C'_1, \cdots$ of Alg from the initial configuration C'_0 such that $s(a'_i, C'_0) = s(a_i, C_t)$ holds for any i $(1 \leq i \leq n/2)$. Since all agents are *red* at C'_0, some agents must change their colors to reach a stable configuration. This implies that, after C_t in execution E, agents change their colors if they interact similarly to E'. This is a contradiction. ◀

5 Conclusion

In this paper, we completely clarify constant-space solvability of the uniform bipartition problem and minimum requirement of agent space under various assumptions. This paper leaves many open problems:

- In the model of a single BS, how many states are necessary and sufficient to develop a uniform bipartition protocol with arbitrary initial states under weak fairness?
- Is it possible to extend our results to the uniform k-partition problem, which divides a population into k groups of the same size. Is it possible to construct a general protocol to solve the uniform k-partition problem? How many states are required to solve the problem?
- What is the relation between the uniform bipartition problem and other problems such as counting, leader election, and majority?
- What is the time complexity of the uniform bipartition problem under probabilistic fairness? The uniform bipartition problem has a close relationship to computation of function $f(n) = n/2$. The time complexity of $n/2$ computation has been studied in [5, 13]. Is it possible to derive the time complexity of the uniform bipartition problem from the results?

References

1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Proc. of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2560–2579, 2017.

2 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proc. of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491, 2015.

3 Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *Proc. of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 47–56, 2015.

4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006.

5 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

6 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.

7 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

8 Dana Angluin, James Aspnes, Michael J Fischer, and Hong Jiang. Self-stabilizing population protocols. In *International Conference On Principles Of Distributed Systems*, pages 103–117. Springer, 2005.

9 James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohier. Time and space optimal counting in population protocols. In *Proc. of International Conference on Principles of Distributed Systems*, pages 13:1–13:17, 2016.

10 James Aspnes and Eric Ruppert. An introduction to population protocols. In *Middleware for Network Eccentric and Mobile Applications*, pages 97–120, 2009.

11 Joffroy Beauquier, Janna Burman, Simon Claviere, and Devan Sohier. Space-optimal counting in population protocols. In *Proc. of International Symposium on Distributed Computing*, pages 631–646, 2015.

12 Joffroy Beauquier, Julien Clement, Stephane Messika, Laurent Rosaz, and Brigitte Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *Proc. of International Symposium on Distributed Computing*, pages 63–76, 2007.

13 Amanda Belleville, David Doty, and David Soloveichik. Hardness of computing and approximating predicates and functions with leaderless population protocols. In *Proc. of 44th International Colloquium on Automata, Languages, and Programming*, pages 141:1–141:14, 2017.

14 Olivier Bournez, Jérémie Chalopin, Johanne Cohen, and Xavier Koegler. Playing with population protocols. In *Proc. of the International Workshop on the Complexity of Simple Programs*, pages 3–15, 2008.

15 Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.

16 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die: Making population protocols fault-tolerant. *Distributed Computing in Sensor Systems*, pages 51–66, 2006.

17 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. In *Proc. of International Symposium on Distributed Computing*, pages 602–616, 2015.

18 Leszek Gasieniec, David Hamilton, Russell Martin, Paul G Spirakis, and Grzegorz Stachowiak. Deterministic population protocols for exact majority and plurality. In *Proc. of International Conference on Principles of Distributed Systems*, pages 14:1–14:14, 2016.

19 Taisuke Izumi. On space and time complexity of loosely-stabilizing leader election. In *Proc. of International Colloquium on Structural Information and Communication Complexity*, pages 299–312, 2015.

20 Tomoko Izumi, Keigo Kinpara, Taisuke Izumi, and Koichi Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theoretical Computer Science*, 552:99–108, 2014.

21 Anissa Lamani and Masafumi Yamashita. Realization of periodic functions by self-stabilizing population protocols with synchronous handshakes. In *Proc. of International Conference on Theory and Practice of Natural Computing*, pages 21–33, 2016.

22 Satoshi Murata, Akihiko Konagaya, Satoshi Kobayashi, Hirohide Saito, and Masami Hagiya. Molecular robotics: A new paradigm for artifacts. *New Generation Computing*, 31(1):27–45, 2013.

23 Yuichi Sudo, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. The same speed timer in population protocols. In *Proc. of International Conference on Distributed Computing Systems*, pages 252–261, 2016.

24 Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theoretical Computer Science*, 444:100–112, 2012.

25 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election on arbitrary graphs in population protocols without identifiers nor random numbers. In *Proc. of International Conference on Principles of Distributed Systems*, pages 14:1–14:16, 2015.

Fast Detection of Stable and Count Predicates in Parallel Computations

Himanshu Chauhan[1] **and Vijay K. Garg**[2]

1 **University of Texas at Austin, USA**
himanshu@utexas.edu
2 **University of Texas at Austin, USA**
garg@ece.utexas.edu

Abstract

Enumerating all consistent states of a parallel computation that satisfy a given predicate is an important problem in debugging and verification of parallel programs. We give a fast algorithm to enumerate all consistent states of a parallel computation that satisfy a stable predicate. In addition, we define a new category of global predicates called *count* predicates and give an algorithm to enumerate all consistent states (of the computation) that satisfy it. All existing predicate detection algorithms, such as BFS, DFS and Lex algorithms, do not exploit the knowledge about the nature of the predicates, and thus may visit all global states of the computation in the worst case. In comparison, our algorithms only visit the states that satisfy the given predicate, and thus take time and space that is a polynomial function of the number of states of interest. In doing so, they provide a significant reduction – exponential in many cases – in time complexities in comparison to existing algorithms.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Algorithms, Theory, Predicate Detection, Parallel Programs

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.20

1 Introduction

Predicate detection [18, 11] is a powerful technique for verifying parallel programs. It allows inference based analysis to check many possible system states based on a single execution of the program. It involves three key steps: (a) obtaining an execution *trace* of the program, (b) modeling this trace as a partial order, and (c) checking all possible states of the model that are consistent with the partial order against a *predicate* that encodes the violation of any constraint or invariant. A large body of work uses this approach to verify distributed applications, as well as to detect data-races and other concurrency related bugs in shared memory parallel programs [10, 14, 21, 25].

In many debugging/analysis applications, we may be interested in analyzing each consistent global state – often called a *consistent cut* – of a parallel program that satisfies a given predicate. For example, while debugging an implementation of the Paxos [24] algorithm, a programmer might only be interested in analyzing consistent cuts when all the *promise* messages of a particular round have been delivered. Another scenario is when a programmer knows that her program exhibits a bug only after the system has executed a certain number of, let us say k, events. For these two scenarios, our predicate definitions are: $B = $ *all promises have been delivered*, and $B = $ *at least k events have been executed*. Both of these predicates fall under the category of *stable predicates*. A stable predicate is a predicate that remains true once it becomes true. In addition, some predicates are defined on the count of

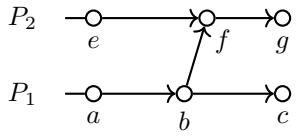


Figure 1 Computation on two processes with six events.

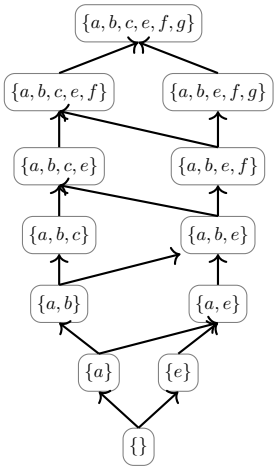


Figure 2 Lattice of Consistent Cuts for Figure 1.

some specific types of events in the system. We call such global predicates *count predicates*. This category of predicates encodes many useful conditions for debugging/verification of parallel programs. For example, $B =$ *exactly two promise messages have been received* is a count predicate.

Let us call the partial order model of the execution trace a *computation*. If we are interested in enumerating all possible consistent cuts of a computation that satisfy a global predicate B that is of either a stable or a count predicate, then we currently only have one choice: traverse all the cuts using existing traversal algorithms (such as BFS [11], DFS [1], and Lex [16, 17] and check which ones satisfy B. This is generally wasteful because we traverse many more cuts than needed – especially if the subset of cuts satisfying B is relatively small. For example, consider the computation in Figure 1, and the predicate $B =$ *at least* 4 *events have been executed*. Figure 2 shows all the consistent cuts of the computation as a *distributive lattice* using the vector clock notation. There are five such cuts in which at least four events have been executed. Using the BFS, DFS, or Lex traversal algorithms, however, we will have to visit all the twelve cuts to find these five.

We present the first algorithms to efficiently enumerate the subset of consistent cuts that satisfy stable or count predicates without enumerating other consistent cuts that do not satisfy them. Our algorithms take time and space that is a polynomial function of the number of consistent cuts of interest. Thus, when compared to existing algorithms for enumerating such consistent cuts, the time complexities of our algorithms are significantly – exponentially for many cases – better.

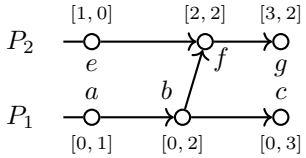


Figure 3 Vector clocks of events for computation of Figure 1.

2 Background

We model a computation $P = (E, \rightarrow)$ on n processes $\{P_1, P_2, \ldots, P_n\}$ as a partial order on the set of events, E. The events are ordered by Lamport's *happened-before* ($\rightarrow$) relation [23]. This partially ordered set (poset) of events is generally partitioned into chains:

▶ **Definition 1** (Chain Partition). A chain partition of a poset places every element of the poset on a chain that is totally ordered. Formally, if α is a chain partition of poset $P = (E, \rightarrow)$ then α maps every event to a natural number such that

$$\forall x, y \in E : \alpha(x) = \alpha(y) \Rightarrow (x \rightarrow y) \vee (y \rightarrow x).$$

Generally, a computation on n processes is partitioned into n chains such that the events executed by process P_i $(1 \leq i \leq n)$ are placed on i^{th} chain.

Mattern [26] and Fidge [13] proposed *vector clocks*, an approach for time-stamping events in a computation such that the happened-before relation can be tracked. For a program on n processes, each event's vector clock is a n-length vector of integers. Note that vector clocks are dependent on chain partition of the poset that models the computation. For an event e, we denote $e.V$ as its vector clock. Throughout this paper, we use the following representation for interpreting chain partitions and vector clocks:

- If there are n chains in the chain partition of the computation, then the lowest chain (process) is always numbered 1, the second lowest chain is numbered 2, and so on, with the highest chain being numbered n.
- A vector clock on n chains is represented as a n-length vector: $[e_n, e_{n-1}, ..., e_i, ..., e_2, e_1]$ such that the rightmost (first from right) entry denotes the number of events executed on P_1, second from right entry denotes events executed on P_2, and so on, such that the left most (n^{th} from right) entry holds the number of events executed on P_n.

Hence, if event e was executed on process P_i, then $e.V[i]$ is e's index (starting from 1) on P_i. Also, for any event f in the computation: $e \rightarrow f \Leftrightarrow \forall j : e.V[j] \leq f.V[j] \wedge \exists k : e.V[k] < f.V[k]$. A pair of events, e and f, is concurrent iff $e \not\rightarrow f \wedge f \not\rightarrow e$. We denote this relation by $e||f$. Note that concurrent events must occur on separate processes. Figure 3 shows the corresponding vector clocks of the computation shown in Figure 1. Event b is the second event on process P_1, and thus using the notation described above its vector clock is $[0, 2]$. Event g is the third event on P_2, but it is preceded by f, which in turn is causally dependent on b on P_1, and thus the vector clock of f is $[3, 2]$.

▶ **Definition 2** (Consistent Cut). Given a computation $(E, \rightarrow)$, a subset of events $C \subseteq E$ forms a *consistent cut* if C contains an event e only if it contains all events that happened-before e. Formally, $e \in C \wedge f \rightarrow e \implies f \in C$.

A consistent cut captures the notion of a global state of the system at some point during its execution [3]. We use the term *cut* for a possible global state of the computation which may or may not be consistent. When the global state is consistent, we use the term consistent cut.

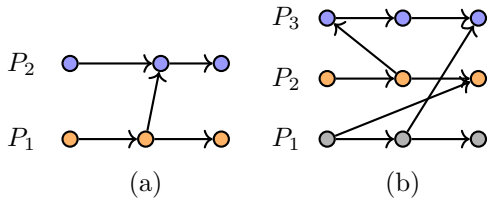


Figure 4 Posets in Uniflow Partitions.

Consider the computation shown in Fig 1. The subset of events $\{a, b, e\}$ forms a consistent cut, whereas the subset $\{a, e, f\}$ does not; because $b \rightarrow f$ (b happened-before f) but b is not included in the subset.

Vector Clock Notation of Cuts. So far we have described how vector clocks can be used to time-stamp events in the computation. We also use them to represent cuts of the computation. If the computation is partitioned into n chains, then for any cut G, its vector clock is a n-length vector such that $G[i]$ denotes the number of events from P_i included in G. Note that in our vector clock representation the events from P_i are at the i^{th} index from the right.

For example, consider the state of the computation in Figure 1 when P_1 has executed events a and b, and P_2 has only executed event e. The consistent cut for this state, $\{a, b, e\}$, is represented by $[1, 2]$. Note that cut $[2, 1]$ is not consistent, as it indicates execution of f on P_2 without b being executed on P_1.

▶ **Definition 3** (Lexical Order on Consistent Cuts)**.** Given any chain partition of poset P that partitions it into n chains, we define a total order called *lexical order* on all consistent cuts of P as follows. Let G and H be any two consistent cuts of P. Then, $G <_l H \equiv \exists k : (G[k] < H[k]) \wedge (\forall i : n \geq i > k : G[i] = H[i])$

▶ **Theorem 4.** *[12, 26] Let $\mathcal{C}(E)$ denote the set of all consistent cuts of a computation $(E, \rightarrow)$. $\mathcal{C}(E)$ forms a finite distributive lattice under the relation $\subseteq$.*

Figure 9 in Appendix A, shows the twelve consistent cuts of the computation in Figure 1 in set notation and their corresponding vector clock notation.

Rank of a Cut. Given a cut G, we define $rank(G) = \sum G[i]$. The rank of a cut corresponds to the total number of events, across all processes, that have been executed to reach the cut.

3 Uniflow Chain Partition

We now discuss a special chain partition of a poset called *uniflow chain partition*. A uniflow partition of a poset P is its partition into n_u chains $\{\mu_i \mid 1 \leq i \leq n_u\}$ such that no element in a higher numbered chain is smaller than any element in lower numbered chain; that is if any element e is placed on a chain i then all elements smaller than e must be placed on chains numbered lower than i. For poset P, chain partition μ is uniflow if

$$\forall x, y \in P : \mu(x) < \mu(y) \Rightarrow (y \not\rightarrow x) \tag{1}$$

Visually, in a uniflow chain partition all the edges between separate chains always point upwards. Figure 4 shows two posets with uniflow partitions. Whereas Figure 5 shows two posets with partitions that do not satisfy the uniflow property. The poset in Figure 5a can

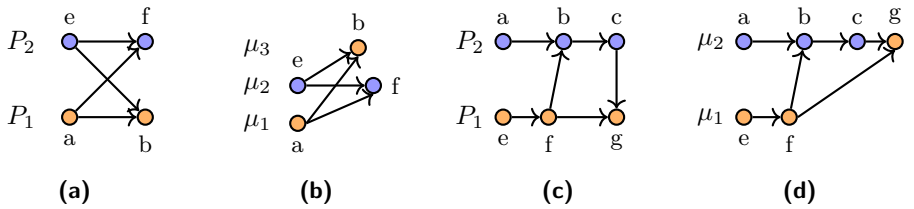


Figure 5 Posets in (a) and (c) are not in uniflow partition: but (b) and (d) respectively are their equivalent uniflow partitions.

be transformed into a uniflow partition of three chains as shown in Figure 5b. Similarly, Figure 5c can be transformed into a uniflow partition of two chains shown in Figure 5d. Observe that:

▶ **Lemma 5.** *Every poset has at least one uniflow chain partition.*

Proof. Any total order that is an extension of the poset is a uniflow chain partition in which each element is a chain by itself. In this trivial uniflow chain partition the number of chains is equal to the number of elements in the poset. ◀

For any poset P, the number of chains in any of its uniflow partition is always less than or equal to $|P|$ (the number of elements in poset). Let us now focus on finding a uniflow chain partition of our poset model of a parallel computation.

We define a total order, called *uniflow order*, on the events of the computation based on its uniflow chain partition. Recall from Equation 1 that for any event e, $\mu(e)$ denotes its chain number in μ. Let $pos(e)$ denote the index of event e on chain $\mu(e)$. Note that a chain is totally ordered, and thus for any two events on the same chain one event's index will be greater than the other's.

▶ **Definition 6** (Uniflow Order on Events, $<_u$)**.** Let μ be uniflow chain partition of a computation $P = (E, \rightarrow)$ that partitions it into n_u chains. We define a total order called *uniflow order* on the set of events E as follows. Let e and f be any two events in E. Then, $e <_u f \equiv (\mu(e) < \mu(f)) \vee (\mu(e) = \mu(f) \wedge pos(e) < pos(f))$

For example, in Figure 5b we have $a <_u e$ as $\mu(a) = 1$ and $\mu(e) = 2$; and $e <_u f$ as $\mu(e) = \mu(f) = 2$, $pos(e) = 1$ and $pos(f) = 2$.

The problem of finding a uniflow chain partition is a direct extension of finding the *jump number* of a poset [9, 2, 32]. Multiple algorithms have been proposed to find the jump number of a poset; which in turn arrange the poset in a uniflow chain partition. Finding an optimal (smallest number of chains) uniflow chain partition of a poset is a hard problem [9, 2]. Bianco et al. [2] present a heuristic algorithm to find a uniflow partition, and show in their experimental evaluation that in most of the cases the resulting partitions are relatively close to optimal. We use the online algorithm given in [7] to find a uniflow partition for a computation.

Recall that for the vector clock notation of a cut G, $G[i]$ denotes the number of events included from chain i. The vector clocks of events, and cuts, of a computation are dependent on the underlying chain partition, and hence we re-generate the vector clocks of the events for the uniflow partition. This is a simple task using existing vector clock implementation techniques, and we omit these details.

Note that the set of consistent cuts of a computation remains the same irrespective of the chain partition used. Hence, if the computation's uniflow partition is different from its

original chain partition, we re-map the vector clock of consistent cuts in uniflow partition to the vector clocks of cuts in original partition. For the details of this step, we refer the reader to [7].

The structure of uniflow chain partitions can be used for efficiently obtaining bigger consistent cuts. After we find a uniflow chain partition of a computation, and regenerate the vector clocks of events as per this partition, we have the following result.

▶ **Lemma 7** (Uniflow Cuts Lemma). *Let P be a poset with a uniflow chain partition $\{\mu_i \mid 1 \leq i \leq n_u\}$, and G be a consistent cut of P. Then any $H_k \subseteq P$ for $1 \leq k \leq n_u$ is also a consistent cut of P if it satisfies:*

$$\forall i : k < i \leq n_u : H_k[i] = G[i], \text{and}$$

$$\forall i : 1 \leq i \leq k : H_k[i] = |\mu_i|.$$

Proof. Using Equation 1, we exploit the structure of uniflow chain partitions: the causal dependencies of any element e lie only on chains that are lower than e's chain. As G is consistent, and H_k contains the same elements as G for the top $(n_u - k)$ chains, all the causal dependencies that need to be satisfied to make H_k have to be on chain k or lower. Hence, including all the elements from all of the lower chains will naturally satisfy all the causal dependencies, and make H_k consistent. ◀

For example, in Figure 4b, consider the cut $G = [1, 2, 1]$[1] that is a consistent cut of the poset. Then, picking $k = 1$, and using Lemma 7 gives us the cut $[1, 2, 3]$ which is consistent; similarly choosing $k = 2$ gives us $[1, 3, 3]$ that is also consistent. Note that the claim may not hold if the chain partition does not have uniflow property. For example, in Figure 5c, $G = [2, 2]$ is a consistent cut. The chain partition, however, is not uniflow and thus applying Lemma 7 with $k = 1$ gives us $[2, 3]$ which is not a consistent cut as it includes the third event on P_1, but not its causal dependency – the third event on P_2.

We now define the notion of a *base cut*: a consistent cut that is formed by including events from μ in a bottom-up manner.

▶ **Definition 8** (l-Base Cut). Let G be a consistent cut of a computation $P =(E, \rightarrow)$ with uniflow partition μ. Then, we call G a *l-base cut* if $\forall j \leq l : G[j] = |(\mu_j)|$

Thus, in a l-base cut we must include all the events from each chain that is same or lower than μ_l in the uniflow partition μ. In Figure 5b, $\{a, e, b\}$ (or $[1, 1, 1]$ in its vector clock notation) is a consistent cut. It is a 1-base cut as it includes all the elements from chain μ_1, but it is not a 2-base cut as it does not include all the events from the chain μ_2.

4 Enumerating Consistent Cuts Satisfying Stable Predicates

A predicate B is stable if once it becomes true it stays true. Some examples of stable predicates are: deadlock, termination, loss of message, at least k events have been executed, and at least k' messages have been sent.

▶ **Definition 9** (Stable Predicate). Let $\mathcal{C}$ be the set of all consistent cuts of a computation. A predicate B defined on $\mathcal{C}$ is called stable if and only if $\forall G, H \in \mathcal{C} : G \subseteq H$ implies that if $B(G)$ is true then $B(H)$ is also true.

[1] Recall that in our vector clock notation ith entry from the right in the vector clock represents the events included from ith chain from the bottom in the uniflow chain partition.

Algorithm 1 EnumerateStable$((E, \rightarrow), B)$.

Input: Computation $(E, \rightarrow)$ in its uniflow chain partition μ, B: a stable predicate
Output: Enumerate all consistent cuts satisfying B.

```
1: G = GetMinCut(B, {}) // find the lexically smallest consistent cut satisfying B
2: while G ≠ null do
3:     enumerate(G) // enumerate the cut
4:     G = GetMinCut(B, G) // find the next lexically smallest consistent cut >_l G satisfying B
```

Algorithm 2 GetMinCut(B, G).

Input: B: a stable predicate, G: a consistent cut
Output: lexically smallest consistent cut $>_l G$ that satisfies B

```
1: ⟨H, c⟩ = GetBiggerBaseCut(B, G)
2: return BackwardPass(B, c − 1, H)
```

Thus, for any stable predicate B the lattice of consistent cuts can be split in two parts using a boundary: every consistent cut higher than the boundary satisfies B, and no consistent cut lower than the boundary satisfies B. Figure 10 (in Appendix, page 18) presents a visualization for this concept. Our goal is to enumerate all consistent cuts of a computation $P =(E, \rightarrow)$ that satisfy a stable predicate B. Note that if the empty cut, $\{\}$ satisfies B, then by the stability property of B all the consistent cuts of the computation satisfy B. In this case, the problem is equivalent to traversing all the consistent cuts of a computation. We can use a fast traversal algorithm such as QuickLex [5] to do so. We now focus on the non-trivial case, and present our algorithm that enumerates only the consistent cuts that satisfy B, and does not enumerate the remaining parts of the lattice of consistent cuts.

Recall that $\mathcal{C}(E)$ represents the set of all consistent cuts of the computation $P =(E, \rightarrow)$. Let $S_B \subset \mathcal{C}(E)$ be the set of all consistent cuts of P that satisfy a stable predicate B. We use P's uniflow partition μ to enumerate them in their lexical order based on the uniflow partition. Let G and H be two consistent cuts of P, then applying the definition of lexical order (Definition 3) over n_u chains, we get $G <_l H \equiv \exists k : (G[k] < H[k]) \wedge (\forall i : n_u \geq i > k : G[i] = H[i])$.

We use the EnumerateStable routine in Algorithm 1 for this enumeration. We first find the lexically smallest consistent cut G that satisfies B. We then find the next cut that is lexically greater than G and satisfies B, and repeat the process after re-assigning G to this cut. We stop when no such lexically greater cut satisfying B is found.

Given a consistent cut G, and a stable predicate B, we use the GetMinCut routine in Algorithm 2 to find the lexically smallest cut that is greater than G and satisfies B. We use two sub-routines for this task: GetBiggerBaseCut and BackwardPass.

The GetBiggerBaseCut routine in Algorithm 3 takes a consistent cut, G, and returns a pair: the first entry is the lexically smallest l-base cut (Definition 8) H lexically greater than G that satisfies B, and the second entry is the chain number from which we added the last event to H before returning the result. If no such cut H can be found, then we return $\langle \mathsf{null}, -1 \rangle$. We start by copying G into H, and from the lowest chain, $i = 1$, add events to H that are not included in it. Each time we add an event e (not already present in H) to H, we form a bigger consistent cut, and then check if this H satisfies B. Note that we move from lower chains to higher, and by the property of uniflow chain partition, we know that adding events in this order will not violate any causal dependencies and keep the cut consistent. At the first instance of finding a bigger cut that satisfies B, we stop and return the pair $\langle H, i \rangle$, where i is the chain number in μ on which we found e. If we consume all the events from a

(a) Computation. **(b)** Uniflow Partition.

Figure 6 A computation on two processes in: (a) its original non-uniflow partition, (b) equivalent uniflow partition.

Algorithm 3 GetBiggerBaseCut(B, G).

Input: B: a stable predicate, G: a consistent cut
Output: pair $\langle H, i\rangle$: H is the smallest base cut that is lexically greater than G and satisfies B, i is the chain number in μ from which we added the last event to H.

```
1: H = G
2: for (i = 1; i ≤ n_u; i = i + 1) do // go from lowest chain to highest
3:   j = index of the smallest event on chain μ_i that is not included in H
4:   for (; j ≤ |μ_i|; j = j + 1) do // use events on chain i not included in G
5:     H = H ∪ {μ_i[j]} // add event to cut H
6:     // H is guaranteed to be lexically greater than G now
7:     if B(H) then // if H satisfies B
8:       return ⟨H, i⟩ // return H and chain number of the event
9: return ⟨null,-1⟩ // no cut lexically greater than G and satisfying B was found
```

chain, we move to the chain immediately above and repeat this process.

Let us illustrate the execution with an example. Consider the computation in Figure 6b and the predicate B=P_2 *has executed two or more events*, and the call GetBiggerBaseCut $(B, \{c\})$. We use the uniflow partition, and starting at μ_1, with $H = G = \{c\}$, we add the first and only event of this chain, a, to H and get $\{a, c\}$ that is greater than G but does not satisfy B, as a was executed on P_1 in the computation. We now jump to chain μ_2, and find the first event on μ_2 that is not included in H. This event is b, the second event on chain. We add it to H and get $H = \{a, b, c\}$ that still does not satisfy B. We now move to the third chain, and add its only event d to H. We now have $H = \{a, b, c, d\}$ and it satisfies B. We return $\langle H = \{a, b, c, d\}, i = 3\rangle$.

The BackwardPass routine (in Algorithm 4) takes three arguments: a stable predicate B, a chain number *start*, and a consistent cut G that satisfies B. It returns a consistent cut H such that H satisfies B, and H is the lexically smallest member of the set: $\{G' \subseteq G : G'[j] = G[j], start + 1 \leq j \leq n_u\}$. Thus, H is the lexically smallest consistent cut $H \leq_l G$ that satisfies B such that H and G include the same set of events from chains $start + 1$ and higher. Note that whenever $start = n_u$, we have $start + 1 > n_u$, and the routine returns without changing the passed cut. We start from the given chain number and traverse backwards on it removing the events as long as the resulting cut continues to satisfy B. If removing an event will cause the cut to become inconsistent or not satisfy B, we do not remove the event and move to the chain immediately below. Consider the computation in Figure 6b and the predicate B=P_2 *has executed two or more events*, and the call BackwardPass $(B, 2, \{a, b, c, d\})$. We start at chain $i = 2$, and remove the last event on this chain, b, from H, to get $K = \{a, c, d\}$. This cut satisfies B as it has two events c and d that were executed on P_2. We now update $H = K = \{a, c, d\}$. We then try to remove c the first event on chain μ_2 from H, but get the cut $K = \{a, d\}$ that is not consistent – d's causal

Algorithm 4 BackwardPass$(B, start, G)$.

Input: B: a stable predicate, $start$: a chain number (from μ) such that $0 < start < n_u$, G: a base cut that satisfies B.
Output: H: Lexically smallest consistent cut $\leq G$ that satisfies B and has $H[k] = G[k]$ for $start + 1 \leq k \leq n_u$.

```
1:  H = G
2:  for (j = start; j ≥ 1; j = j − 1) do // iterate from start argument chain to lower chains
3:    for (e = H[j]; e ≥ 1; e = e − 1) do // from last event on chain to first
4:      K = H \ {μ_j[e]} // remove event from cut
5:      if K is inconsistent then // removing the event violated consistency
6:        break // break inner loop on events to move to the lower chain
7:      // K must be consistent now
8:      if B(K) then // K is consistent, smaller than G, and satisfies B
9:        H = K // update H to this cut
10: return H
```

dependency c is not included in this cut. Hence, H is not changed, and kept as $\{a, c, d\}$. We now move to the lower chain μ_1. We again cannot remove the only event from this chain (event a) as it will make the cut inconsistent. We now have exhausted all the chains, and thus at the end return $H = \{a, c, d\}$ which is lexically smaller than $G = \{a, b, c, d\}$ and satisfies B.

For the computation in Figure 6b and the predicate B=P_2 *has executed two or more events*, let us find the lexically smallest cut that satisfies B. We use the GetMinCut routine, and since we are interested in finding the lexically smallest cut, we start with $G = \{\}$. Calling GetBiggerBaseCut $(B, \{\})$ returns $\langle H = \{a, b, c, d\}, i = 3\rangle$ as shown earlier. Now calling BackwardPass $(B, 2, \{a, b, c, d\})$ returns $\{a, c, d\}$. This is the lexically smallest cut of the computation that satisfies B.

Let us now go through a run of EnumerateStable routine. For the computation in Figure 6b and the predicate B=P_2 *has executed two or more events*, we have already seen that lexically smallest cut that satisfies B is $\{a, c, d\}$. We enumerate this cut at line 3 (in Algorithm 1) and then call GetMinCut $(B, \{a, c, d\})$. This in turn will first call GetBiggerBaseCut $(B, \{a, c, d\})$, and the result is $\langle H = \{a, b, c, d\}, i = 2\rangle$. The second call (in GetMinCut) is BackwardPass $(B, 1, \{a, b, c, d\})$ that returns $G = \{a, b, c, d\}$, and we enumerate it. The next call of GetMinCut $(B, \{a, b, c, d\})$ will return null as there is no cut greater than $\{a, b, c, d\}$. Hence, the loop will now terminate, and we have enumerated all the cuts that satisfy B.

We present the proof of correctness for these algorithms in this paper's extended version [6]. The routines presented here can be implemented without the regeneration of vector clocks for uniflow partition. We can also optimize them for improved runtime by using some additional space. We present these implementation details, and their optimizations in Appendix B.

5 Enumerating Consistent Cuts satisfying Count Predicates

Many applications involve analysis of computations based on some specific *type* of events. The type of an event is defined either in the context of the system under consideration, or in the context of the analysis problem. For example, we can categorize events in a message-passing computation in three base types: *send* event, *receive* event, and *local* event. Similarly, in a shared memory parallel computation that uses locks, we can define three

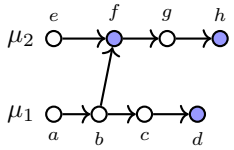


Figure 7 A computation in uniflow partition.

base types: *acquire-lock* event, *release-lock* event, and *thread-local* event. Analyzing such computations may require us to check all consistent cuts that satisfy *counting* conditions on a type of event. For example, we may be interested in analyzing the computation when a certain number of *send* events have occurred, or a certain number of messages have been received. We call such predicates *count predicates*. Count predicates are used in multiple debugging and analysis applications. For example, while debugging an implementation of Paxos [24] algorithm, a programmer might only be interested in analyzing possible system states when kth *propose* message has been sent, or k' *promise* messages have been delivered. Another scenario is when a programmer knows that a program exhibits a bug only after the system has executed a certain number of events. We use the notion of colors to represent types. We assume that by default each event in a computation is colored white. Then, every event of interest is assigned a color where each color represents a type categorization. Note that an event can have only one color, and on assigning a color c to it, we replace its previously assigned color. For example, in the Paxos implementation scenario discussed earlier, we may assign the color blue to all the events that send a propose message, and the color red to all the events that deliver promise messages. We then define the notion of a *view* of a consistent cut with respect to a color:

▶ **Definition 10** ($view(G, c)$)**.** Let each event e of the computation $P = (E, \rightarrow)$ be colored with a color c from the set of colors C. Then for a consistent cut G of P we define $view(G, c)$ as the set of events that are included in G and are colored c.

For example, consider the computation shown in Figure 7. The events in this computation are colored either white or blue. Given the cut $G = \{a, b, e\}$ in this computation, we have $view(G, white) = \{a, b, e\}$, and $view(G, blue) = \{\}$. For $G = \{a, b, c, d, e, f, g\}$, we get $view(G, white) = \{a, b, c, e, g\}$, and $view(G, blue) = \{d, f\}$.

We now use the view with respect to a color to define a count predicate.

▶ **Definition 11** (Count Predicate)**.** Let $P = (E, \rightarrow)$ be a computation, and c be a color from the set of colors C. A predicate B is called a count predicate if it can be written in the form: $|view(G, c)| = k \in \mathbb{N}$, for any consistent cut G of P.

If c is the color used in defining B, then we use the notation $count_B(G) = |view(G, c)|$. Observe that for a count predicate B, we get:

- $count_B(G) \leq rank(G)$.
- If H is a consistent cut such that $G \subseteq H$ then $count_B(H) \geq count_B(G)$.
- If K is a consistent cut such that $G \subset K$ and $count_B(K) > count_B(G)$, then $\exists H : (G \subset H \subseteq K) \ \wedge count_B(H) = count_B(G) + 1$.

Given that B is defined with respect to one color c, for brevity and ease of notation we usually write $view(G)$ for $view(G, c)$ when c is obvious from the context.

We now present an algorithm to enumerate all consistent cuts of a computation $(E, \rightarrow)$ that satisfy a count predicate B. We use the computation's uniflow partition μ for enumerating

Algorithm 5 EnumerateCount$((E, \rightarrow), B)$.

Input: Computation $(E, \rightarrow)$ in its uniflow chain partition μ, B: a count predicate
Output: Enumerate all consistent cuts satisfying B.

```
1: G = GetMinCut(B, {}) // now G is the smallest cut satisfying B
2: while G ≠ null do
3:    EnumSameViewCuts(B, G)
4:    G = GetSuccessor(B, G)
```

Algorithm 6 EnumSameViewCuts(B, G).

Input: B: a count predicate, G: a consistent cut that satisfies B.
Output: Enumerate each consistent cut H that is $\geq_l G$ and satisfies $view(G) == view(H)$.

```
1: enumerate(G)
2: H = G
3: K = G
4: for (i = 1; i ≤ n_u; i = i + 1) do // go from lowest chain to highest
5:    j = index of the first event on chain μ_i that is not included in H
6:    for (; j ≤ |μ_i|; j = j + 1) do // use events not included in G
7:       H = H ∪ {μ_i[j]} // add event to cut
8:       if view(H) == view(G) then // same view
9:          K = H // update cut
10:         enumerate(K)
11:      else // B(H) = false; count_B(H) must have increased
12:         H = K // retain old cut
13:         break // break the inner loop on events; move to the chain above
```

these cuts in their lexical order. Algorithm 5 shows our approach outline. First we find the lexically smallest cut that satisfies B. Given the properties of B, we know that adding new events to any consistent cut G can either increase $count_B(G)$ or keep it same. Thus, using the uniflow chain partition μ we can use the GetMinCut routine from Algorithm 2 to find the lexically smallest cut that satisfies B. This works because the lexically smallest cut that satisfies the count predicate $count_B(G) = k$ is also the lexically smallest cut that satisfies the stable predicate $count_B(G) > k - 1$. We then repeatedly enumerate lexically bigger cuts that satisfy B using two sub-routines: EnumSameViewCuts and GetSuccessor.

EnumSameViewCuts in Algorithm 6 takes two arguments: a count predicate B, and a consistent cut G that satisfies B. It uses the uniflow chain partition μ to enumerate all the consistent cuts that satisfy the predicate and have the same *view* with respect to the color c used to define B. For example, consider the predicate B=*number of blue events is* 1, and the computation in Figure 7. Calling EnumSameViewCuts with $G = \{a, b, e, f\}$ will enumerate three cuts: $\{a, b, e, f\}, \{a, b, e, f, g\}, \{a, b, c, e, f, g\}$ as they have the same *view* – the same blue event f has been executed in all of them. The routine goes from lower chains to higher, and on each chain adds events in their increasing order to the cut. We know from the structure of uniflow chain partition that the resulting cut will be consistent. If it has the same *view*, then we enumerate it. Otherwise, if the *view* is different, by the properties of B we know that adding more events from the same chain will also give a different *view* than the one we seek. Hence, we move to the chain above, and repeat the steps.

Given a consistent cut G that satisfies B, GetSuccessor routine in Algorithm 7 finds a consistent cut H such that H satisfies B and $view(G) \neq view(H)$. For example, suppose $B =$ *number of blue events is* 2. Then for the computation in Figure 7, given $G = \{a, b, c, d, e, f\}$, we have GetSuccessor $(B, G) = \{a, b, e, f, g, h\}$. This is because $view(\{a, b, c, d, e, f\})$ is

Algorithm 7 GetSuccessor(B, G).

Input: B: a count predicate, G: a consistent cut satisfying B
Output: K: lexically smallest consistent cut $>_l G$ that satisfies B and $view(G) \neq view(K)$

```
1: V = view(G)
2: r = count_B(G)
3: K = G  // Create a copy of G in K
4: for (i = 1; i ≤ n_u; i++) do  // lower chains to higher
5:   ind = index of the first event on chain μ_i that is not included in K
6:   for (; ind ≤ |μ_i|; ind = ind + 1) do // move forward on chain
7:     K = K ∪ {μ_i[ind]} // add event to cut
8:     if view(K) ≠ V then // K is lexically greater than G and has a different view than G
9:       for (j = i − 1; j > 0; j − −) do // first reset lower chains
10:        remove all elements on μ_j from K
11:      //K may not be consistent: fix causual dependencies on all lower chains
12:      for (j = i + 1; j ≤ n_u; j + +) do
13:        for (k = i − 1; k > 0; k − −) do
14:          S = causal dependencies of events from chain μ_j on chain μ_k
15:          K = K ∪ S
16:      //K is a consistent cut now, and view(K) ≠ view(G)
17:      if B(K) == true then
18:        return K // K satisfies B, and is the successor cut we want
19:      if count_B(K) < r then // K can be used to construct the lexically bigger cut that satisfies B
20:        return GetMinCut(B, K)
21: return null  // could not find a candidate cut
```

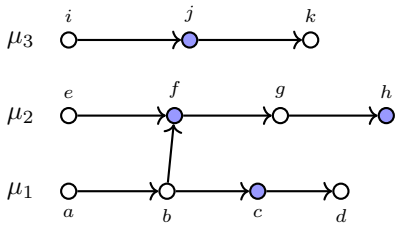


Figure 8 A computation in uniflow partition.

the set with two blue events: $\{d, f\}$. The next lexically bigger consistent cut that has two blue events and has a different *view* is the cut $\{a, b, e, f, g, h\}$ with two blue events: f and h.

In this routine, we start at the lowest chain in a uniflow poset, and if possible increment the cut by one event on this chain. If the new cut has the same *view*, we move on to the next event. When we encounter an event whose addition changes the *view* of the resulting cut K, we reset the entries on lower chains, and then make K consistent by satisfying all the causal dependencies. Note that at this point $view(K)$ is guaranteed to be different than $view(G)$. However, K may not satisfy B as it may have a lower $count_B$. If that is the case, we make $count_B(K) == count_B(G)$ by calling the GetMinCut routine to find lexically smallest cut that is greater than K and satisfies B. If we have tried all chains and did not find a suitable cut, then G is the largest consistent cut satisfying B and we return null.

Consider the computation in Figure 8 which is in a uniflow partition. Given the predicate B = *number of blue events is* 2, and consistent cut $G = \{a, b, c, d, e, f\}$ that satisfies B, consider the call of GetSuccessor (B, G). We find $V = view(G) = \{c, f\}$, and $r = count_B(G) = 2$,

and create $K = G$. We start from the bottom chain μ_1 but there is no event in μ_1 that is not included in K. We move on to μ_2 and find the next event not in K: event g. We add it to K at line 7, to make $K = \{a, b, c, d, e, f, g\}$, which is bigger than G but $view(K) == V$ as g is not a blue event. We then move on to the next event in μ_2 which is h. Adding it to K makes $K = \{a, b, c, d, e, f, g, h\}$. Now K is bigger than G and $view(K) = \{c, f, h\}$ which is different than V. We now remove all the events (lines 9–10) from lower chain μ_1, and get $K = \{e, f, g, h\}$. This cut is not consistent, and we make it consistent by executing lines 12–15 and add all the causal dependencies required: $\{a, b\}$. We now have $K = \{a, b, e, f, g, h\}$. At line 17, we get $count_B(K)$ which is 2; thus we have our result and we return this K. Hence, GetSuccessor $(B, G) = \{a, b, e, f, g, h\}$ whose *view* is $\{f, h\}$. If we call GetSuccessor $(B, \{a, b, e, f, g, h\})$, we get $\{a, b, c, i, j\}$ whose *view* is $\{c, j\}$.

We present the proof of correctness for these algorithms in this paper's extended version [6]. The algorithms presented in this section can be implemented without regeneration of vector clocks for uniflow partition. In addition, we can optimize them for improved runtime performance using some additional space. We discuss these implementation details in Appendix B.

6 Complexity Analysis

Consider the computation $P = (E, \rightarrow)$ whose uniflow partition μ has n_u chains. We now present the time and space complexity of the optimized versions of our algorithms for detecting stable and count predicate for P.

Using the optimized implementations discussed in Appendix B.1, we know that computing and storing the vector J requires $\mathcal{O}(n \cdot |E|)$ time and space. This task is only performed once. After computing J, each call to GetBiggerBaseCut takes $\mathcal{O}(n \log |E|)$ time with binary search: there are $\mathcal{O}(\log |E|)$ search iterations, and for each such iteration, we require $\mathcal{O}(n)$ time to check if the consistent cut under consideration satisfies the predicate. Similarly, using the optimized implementation from Appendix B.2, BackwardPass takes $\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ time, where $m = \max_{1 \leq j \leq n_u} |\mu_j|$, in the worst case. Hence, getting a consistent cut result from GetMinCut in the representation corresponding to original chain partition takes $\mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n)$ time in the worst case.

Based on this, we can state that for a stable predicate B enumerating all consistent cuts of $P = (E, \rightarrow)$ that satisfy B takes $\mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n)$ time per cut.

Let us now analyze the EnumSameViewCuts routine. Given a cut G, the routine adds events not already present in G to form bigger cuts, and then checks if the cut satisfies the predicate B. There are at $|E - G|$ events that are not present in G. Hence, in the worst case the two for loops at lines 4 and 6 perform $\mathcal{O}(|E - G|)$ iterations in combination. Each time we form a bigger cut by adding an event, we check if the *view* of the cuts remains the same (at line 8). Finding $view(H)$ requires $\mathcal{O}(n)$ time. Thus, EnumSameViewCuts takes $\mathcal{O}(n \cdot |E - G|)$ in the worst case.

We now analyze the optimized version of GetSuccessor routine. Recall that with the projection based optimization, we first call the ComputeProjections routine that takes $\mathcal{O}(n \cdot n_u)$ time. We need $\mathcal{O}(n \cdot n_u)$ space to store the computed projections. We then iterate over n_u chains, and perform $\mathcal{O}(n)$ work in finding $viewK$ and then $\mathcal{O}(n)$ work in taking the component-wise maximum of $proj[i-1]$ and the vector clock of event being included. Thus, in the worst case we perform $\mathcal{O}(n \cdot n_u)$ work before returning a result. Note that, we may call GetMinCut routine at the end to return the correct result. As per our earlier analysis, that requires additional $\mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n)$ time. Hence, in the worst

Table 1 Space complexities of algorithms for detecting a stable or count predicate in the lattice of consistent cuts; here $m = \frac{|E|}{n}$.

Algorithm	Space Required
BFS [11]	$\mathcal{O}(\frac{m^{n-1}}{n})$
DFS [1]	$\mathcal{O}(\|E\|)$
Lex [17]	$\mathcal{O}(n)$
QuickLex [5]	$\mathcal{O}(n)$
This paper*	$\mathcal{O}((n_u + \|E\|) \cdot n)$

Table 2 Time complexities for enumerating all consistent cuts of $\mathcal{C}(E)$ that satisfy a stable predicate B. Here Δ is the maximum in-degree of an event in the computation.

Algorithm	Time
BFS [11]	$\mathcal{O}(n^2 \cdot \|\mathcal{C}(E)\|)$
DFS [1]	$\mathcal{O}(n^2 \cdot \|\mathcal{C}(E)\|)$
Lex [17]	$\mathcal{O}(n^2 \cdot \|\mathcal{C}(E)\|)$
QuickLex [5]	$\mathcal{O}(n \cdot \Delta \cdot \|\mathcal{C}(E)\|)$
This paper*	$\mathcal{O}(n \cdot \|S_B\| \cdot (n_u \cdot n \cdot \log m + \log \|E\|))$

case GetSuccessor takes $\mathcal{O}((n_u \cdot n \cdot \log m + \log |E|) \cdot n)$ time and requires $\mathcal{O}(n \cdot n_u)$ space.

In Table 1, we compare the worst-case space complexities of our optimized algorithm against those of detecting the predicate using the BFS, DFS, and Lex traversal algorithms.

Let $S_B \in \mathcal{C}(E)$ denote the set of consistent cuts that satisfy the stable or count predicate B for the computation $P = (E, \rightarrow)$. Then, based on our analysis we have the following result:

▶ **Theorem 12.** *Enumerating all consistent cuts in S_B takes $\mathcal{O}(f \cdot |S_B|)$ time using the algorithms given in this paper; where f is a polynomial function of $|E|$ and n.*

In comparison, enumerating all the cuts of S_B using the existing algorithms such as BFS, DFS, Lex (or QuickLex) may take $\mathcal{O}(|\mathcal{C}(E)|)$ time in the worst case. Note that the $|\mathcal{C}(E)|$ can be exponentially bigger than $|S_B|$. Table 2 compares the worst-case time complexities of these algorithms to enumerate all consistent cuts in S_B when B is stable.

7 Related Work

We first discuss the algorithms for traversal of cuts in the lattice of consistent cuts of a computation. Cooper and Marzullo [11] gave the first algorithm for global states enumeration which is based on breadth first search (BFS). Let $i(P)$ denote the total number of consistent cuts of a poset P. Cooper-Marzullo algorithm requires $\mathcal{O}(n^2 \cdot i(P))$ time, and exponential space in the size of the input computation. Alagar and Venkatesan [1] presented a depth first algorithm using the notion of global interval which reduces the space complexity to $\mathcal{O}(|E|)$. Steiner [31] gave an algorithm that uses $\mathcal{O}(|E| \cdot i(P))$ time, and Squire [30] further improved the computation time to $\mathcal{O}(log|E| \cdot i(P))$. Pruesse and Ruskey [29] gave the first algorithm that generates global states in a combinatorial Gray code manner. The algorithm uses $\mathcal{O}(|E| \cdot i(P))$ time and can be reduced to $\mathcal{O}(\Delta(P) \cdot i(P))$ time, where $\Delta(P)$ is the in-degree of an event; however, the space grows exponentially in $|E|$. Later, Jegou et al. [22] and Habib et al. [20] improved the space complexity to $\mathcal{O}(n \cdot |E|)$. Ganter [16] presented an

algorithm, which uses the notion of lexical order, and Garg [17] gave the implementation using vector clocks. The lexical algorithm requires $\mathcal{O}(n^2 \cdot i(P))$ time but the algorithm itself is *stateless* and hence requires no additional space besides the poset. Paramount [4] gave a parallel algorithm to traverse this lattice in lexical order, and QuickLex [5] provides an improved implementation for lexical traversal that takes $\mathcal{O}(n \cdot \Delta(P) \cdot i(P))$ time, and $\mathcal{O}(n^2)$ space overall.

For enumerating only the consistent cuts that satisfy a given predicate, computation slicing is an abstraction technique to reduce the state space for model checking a single program trace [27, 28, 8]. But the known efficient algorithms for computation slicing generally require the predicate to be *regular*. Hence, this technique does not apply to stable or count predicates.

8 Conclusion and Applications to Other Fields

The ubiquity of multicore and cloud computing has significantly increased the degree of parallelism in programs. This change has in turn made verification and analysis of large parallel programs even more challenging. For such verification and analysis tasks, the algorithms presented in this paper can provide much faster runtimes in comparison to existing algorithms. In many cases, this reduction in runtime can be exponential, and also allows us allows us to analyze computation with high degree of parallelism with relatively small memory footprint.

Our algorithm for detecting count predicates has a wide-ranging potential scope in analysis of parallel computations. In addition to predicate detection for verifying correctness, it can also be used to analyze logs of distributed protocols such as Paxos, and various distributed systems for performance related analysis. Further optimizations of this algorithm can provide improved runtimes for its implementation which can make it an appealing choice as a lightweight and fast component in online runtime verification systems.

Observe that many useful analysis criterion can be written in the form of stable predicates. For example, if we are interested in analyzing logs of a distributed system to identify causes of a system failure or performance degradation, we can create stable predicates that include either thresholds or upper bounds for performance load factors. By using these predicates, we can then use our algorithm (Algorithm 1) to efficiently find only those system states that are of interest to us without going through the states that came before them. A promising future application of our work is implementation of a system that accepts either a stable or count predicate and returns the set of consistent cuts satisfying it.

The applications of our algorithms extend to the problem of stable marriage [15, 19]. The stable marriage problem involves finding a stable matching of women and men and ensure that there is no pair of woman and man such that they are not married to each other but prefer each other over their matched partners. Many variations of the problem with additional constraints have been studied. Some examples include *man-optimal* or *woman-optimal* matchings, and introducing the notion of *regret*. We can use the algorithms developed in this paper to enumerate matchings that meet a given lower-bound or upper-bound, or any other combination of such criteria on the overall cumulative regret of the matching, or individual regrets of actors.

The notion of consistent cut of a computation, directly maps to the notion of *order ideals* in a lattice. Multiple problems in the field of lattice theory require enumeration of a specific level of order ideals, or a range of levels. Our algorithms can be used to enumerate order ideals of a lattice that satisfy some stable properties without visiting other levels of the

lattice. Our algorithm for enumerating cuts satisfying count predicate can also be used to traversing order ideals of a sub-lattice without visiting ideals outside the sub-lattice. No known algorithm in lattice theory has the ability to perform such traversals without visiting other ideals of the lattice – whose total number can be exponentially bigger than the size of the sub-lattice of interest.

References

1 S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering (TSE)*, 27(8):704–714, AUG 2001.

2 Lucio Bianco, Paolo Dell 'Olmo, and Stefano Giordani. An optimal algorithm to find the jump number of partially ordered sets. *Computational Optimization and Applications*, 8(2):197–210, 1997.

3 K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, FEB 1985.

4 Yen-Jung Chang and Vijay K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 140–149. ACM, 2015.

5 Yen-Jung Chang and Vijay K. Garg. Quicklex: A fast algorithm for consistent global states enumeration of distributed computations. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, pages 25:1–25:17, 2015.

6 Himanshu Chauhan and Vijay K. Garg. Fast detection of stable and counting predicates in parallel computations. *Extended Version*, 2017. URL: `http://users.ece.utexas.edu/~garg/dist/opodis17.pdf`.

7 Himanshu Chauhan and Vijay K. Garg. Space efficient breadth-first and level traversals of consistent global states of parallel programs. In *17th International Conference on Runtime Verification (RV 2017)*, pages 138–154, 2017.

8 Himanshu Chauhan, Vijay K Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate detection. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 101–110. IEEE, 2013.

9 M Chein and M Habib. The jump number of dags and posets: an introduction. *Annals of Discrete Mathematics*, 9:189–194, 1980.

10 Feng Chen, Traian Florin Serbanuta, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.

11 R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

12 B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

13 C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial-Ordering. In K. Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, pages 56–66, FEB 1988.

14 Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

15 David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

16 Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

17 Vijay K. Garg. Enumerating global states of a distributed computation. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 134–139, 2003.

18 Vijay K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

19 Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

20 Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.

21 Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

22 Roland Jegou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In *Proc. of the International Workshop on Structures in Concurrency Theory*, pages 175–189, 1995.

23 L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, JUL 1978.

24 Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

25 Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

26 F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226, 1989.

27 N. Mittal and V. K. Garg. Techniques and Applications of Computation Slicing. *Distributed Computing (DC)*, 17(3):251–277, MAR 2005.

28 N. Mittal, A. Sen, and V. K. Garg. Solving Computation Slicing using Predicate Detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18(12):1700–1713, DEC 2007.

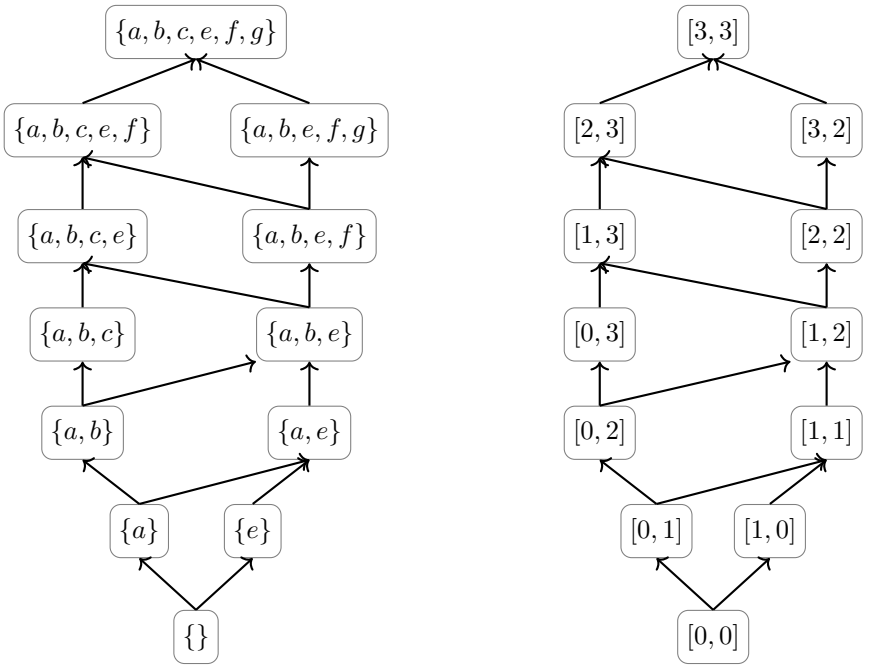
29 Gara Pruesse and Frank Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.

30 Matthew B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

31 George Steiner. An algorithm to generate the ideals of a partial order. *Oper. Res. Lett.*, 5(6):317–320, 1986.

32 Maciej M Sysło. Minimizing the jump number for partially ordered sets: A graph-theoretic approach. *Order*, 1(1):7–19, 1984.

A Visual Illustrations for Lattice of Consistent Cuts



(a) Set of Events Representation. **(b)** Vector Clock Representation.

Figure 9 Lattice of Consistent Cuts for Figure 1.

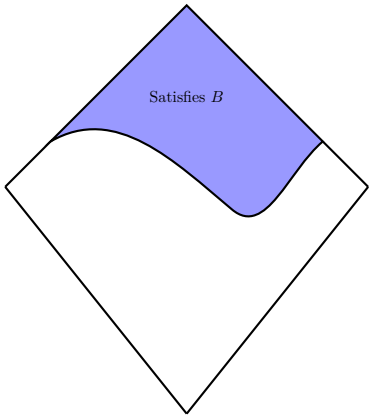
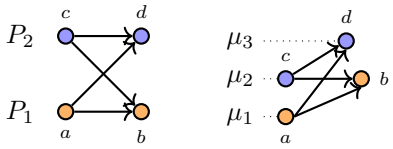


Figure 10 Illustration: Visual representation for some stable predicate B: the cuts in the blue region of the lattice satisfy a stable predicate, and cuts in the white region do not.

B Optimized Implementation

In this section, we discuss optimized implementations of our algorithms for detecting stable and counting predicates.

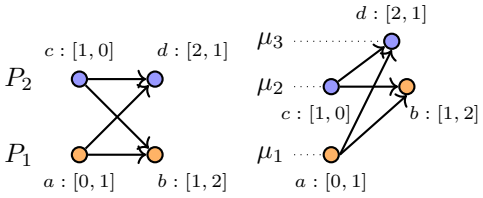
First, note that we do not need to regenerate the vector clocks of the computation for its uniflow chain partition. In implementing our algorithms based on the uniflow chain partition,



$G = \{a, b, c, d\} \implies G_u[0] = 1, G_u[1] = 2, G_u[2] = 1$
$G = \{a, c, d\} \implies G_u[0] = 1, G_u[1] = 1, G_u[2] = 1$
$G = \{a, c, b\} \implies G_u[0] = 1, G_u[1] = 2, G_u[2] = 0$
$G = \{a, c\} \implies G_u[0] = 1, G_u[1] = 1, G_u[2] = 0$
$G = \{a\} \implies G_u[0] = 1, G_u[1] = 0, G_u[2] = 0$

(a) Computation. **(b)** Uniflow Partition. **(c)** G values and their respective G_u vectors.

Figure 11 Illustration: Maintaining indicator vector G_u for a cut G.



$d : [2, 1]$
$b : [1, 2]$
$c : [1, 0]$
$a : [0, 1]$

$J[4] = [2, 2]$
$J[3] = [1, 2]$
$J[2] = [1, 1]$
$J[1] = [0, 1]$

(a) Computation. **(b)** Uniflow Partition. **(c)** Events in Uniflow Order. **(d)** J Vector.

Figure 12 Illustration: Computing J vector for optimizing GetBiggerBaseCut.

μ, we only reposition the events on their respective uniflow chains. There are n_u such chains, and each of them is stored as an array in which whose entries store the original vector clocks, and the state variables for each event. For example, the computation on two processes in Figure 12a is not in uniflow partition. Figure 12b shows its uniflow partition on three chains. Note that we have retained the original vector clocks of the events, and only repositioned them on three chains.

We achieve this by replicating the process described in [7]. In short, we use a vector G_u, called *indicator vector*, of length n_u, to keep track of which event is included in G. In Figure 11, we show an illustration with multiple G cuts, and their respective indicator vectors. Whenever we add an event e from chain μ_i to G we update $G_u[i]$ to the index of e. Thus, finding the index of the first event on chain μ_i not included in G can be implemented as $ind = G_u[i] + 1$, and takes constant time. Given the indicator vector G_u, we can find its equivalent cut G in $\mathcal{O}(n_u + n^2)$ time. For details, we refer the interested reader to Section 4.2 of [7].

B.1 GetBiggerBaseCut

In the GetBiggerBaseCut routine we add events to any cut in increasing uniflow order (Definition 6). We do not skip any event, and only return $\langle H, c\rangle$ when the cut satisfies a predicate B. Given a uniflow chain partition μ, we can optimize the runtime for this routine by using additional $\mathcal{O}(n \cdot |E|)$ space.

The computation $P = (E, \rightarrow)$ on n processes has $|E|$ events, and each event has a vector clock of length n. We first collect and store all the events in the uniflow order. Let J represent the array that stores the vector clocks of events in their increasing uniflow order. Now, for $2 \leq i \leq |E|$ we compute element-wise max of vector clocks in entries $J[i]$ and $J[i-1]$, and store the result in $J[i]$. Thus, for a computation on n processes $J[i]$ and $J[i-1]$ are both vector of length n, and we have:

$$J[i][k] = \max\left(J[i][k], J[i-1][k]\right), \quad 2 \leq i \leq |E|, 1 \leq k \leq n.$$

We can now use this vector J to find the result of GetBiggerBaseCut for any predicate

B. Moreover, given that J will contain entries (vector clocks) in increasing order, we can perform binary search on it to find the result. If a predicate B is stable, we perform the binary search using its evaluation (true or false) on the cuts, and return the smallest entry in J on which B evaluates to true.. If B is a counting predicate, then we use $count_B$ to guide the binary search, and return the smallest entry in J for which $count_B$ matches the requirement in B.

Consider the computation in Figure 12a that has four events, and its uniflow partition in Figure 12b. The increasing order on the vector clocks of all the four events is in Figure 12c. Starting from the bottom (vector $[0, 1]$), and performing the joins, we get J as shown in Figure 12d. Now, given a predicate B that is stable or counting, we can perform the binary search on this J to find the result of GetBiggerBaseCut for this computation.

Computing and storing the vector J requires $\mathcal{O}(n \cdot |E|)$ time and space. After computing J, each call to GetBiggerBaseCut takes $\mathcal{O}(n \cdot \log |E|)$ time with binary search: there are $\mathcal{O}(n \cdot \log |E|)$ iterations, and for each such iteration we take $\mathcal{O}(n \cdot \log |E|)$ time to check the consistent cut satisfies the predicate.

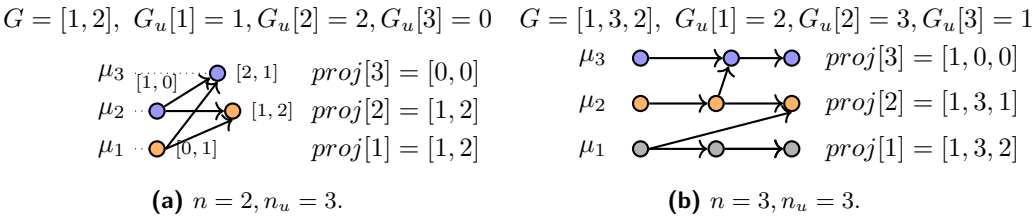
B.2 BackwardPass

In BackwardPass routine, we iterate on chains in top to bottom manner, and try to remove as many events from a cut G from the end of the chain as possible. We only stop removing events from a chain i if G becomes inconsistent or $B(G)$ becomes false on removal. Then, we move to chain $i - 1$. We can exploit the properties of stable and counting predicates, and use binary search, instead of linear search used in Algorithm 4 to remove events on each chain. This is possible possible because for a stable or counting predicate, if removal of an event from a chain makes the predicate become false (from true) then we know that removing any smaller events on that chain will never make it true. Using this implementation, BackwardPass takes $\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ time, where $m = \max_{1 \le j \le n_u} |\mu_j|$, in the worst case. This is because the outer loop on the uniflow chains takes $\mathcal{O}(n_u \cdot n^2 \cdot \log m)$ iterations in the worst case. In the inner body of this loop, we check if removal of an event makes the resulting cut inconsistent, and this check requires $\mathcal{O}(n^2)$ time. There are $\mathcal{O}(\log m)$ search iterations for such an event in the worst case.

B.3 GetSuccessor

We optimize the routine GetSuccessor by replicating the strategy of computing projections as per [7]. Whenever the routine is called, we compute the causal dependencies, called projections, of the input consistent cut on each chain in μ, and store them in a vector called $proj$. We then use this vector to fix the causal dependencies on each chain in $\mathcal{O}(n)$ time (see [7] for details). For this optimization, we require $\mathcal{O}(n_u \cdot n)$ space to store the computed projections, and by using them we can find the result of GetSuccessor in $\mathcal{O}((n_u + \log |E| + n_u \log m) \cdot n)$ time in the worst case. As $\log m > 1$ for most of the computations, we can simplify this bound to $\mathcal{O}((\log |E| + n_u \log m) \cdot n)$.

We illustrate with the computation shown in Figure 13a that was originally on two processes. Suppose we want the lexical successor of $G = [1, 2]$. Then, for each chain, starting from the top, using the vector G_u we compute the projection of events included in G on lower chains. For the consistent cut $G = [1, 2]$, we have $G_u[3] = 0, G_u[2] = 2, G_u[1] = 1$. Hence, on the top-most chain, the projection is empty and we have $proj[3] = [0, 0]$. On chain μ_2, the projection must include the combined vector clocks of events included form chain μ_3, and μ_2. As $G_u[2] = 2$, we take the vector clock of second event on μ_2, and perform a



(a) $n = 2, n_u = 3$. **(b)** $n = 3, n_u = 3$.

Figure 13 Illustration: Projections of cuts on uniflow chains.

element-wise max operation for its entries and $proj[3]$. We thus get $proj[2] = [1, 2]$. We then move to chain μ_1 and find the vector clock of event against entry $G_u[1] = 1$ which is the first event on μ_1, with vector clock $[0, 1]$. We then set $proj[1] = \max(proj[2], [0, 1])$, which is element-wise max of two arrays $[1, 2]$, and $[0, 1]$. Thus, we get $proj[1] = [1, 2]$.

Figure 13b shows another illustration of computing the projection vector $proj$ on a computation with three processes that forms a uniflow chain partition by default.

Fast Distributed Approximation for TAP and 2-Edge-Connectivity*†

Keren Censor-Hillel[1] and Michal Dory[2]

1 **Technion, Department of Computer Science, Haifa, Israel**
ckeren@cs.technion.ac.il

2 **Technion, Department of Computer Science, Haifa, Israel**
smichald@cs.technion.ac.il

Abstract

The *tree augmentation problem (TAP)* is a fundamental network design problem, in which the input is a graph G and a spanning tree T for it, and the goal is to augment T with a minimum set of edges Aug from G, such that $T \cup Aug$ is 2-edge-connected.

TAP has been widely studied in the sequential setting. The best known approximation ratio of 2 for the weighted case dates back to the work of Frederickson and JáJá, SICOMP 1981. Recently, a 3/2-approximation was given for the unweighted case by Kortsarz and Nutov, TALG 2016, and recent breakthroughs by Adjiashvili, SODA 2017, and by Fiorini et al., 2017, give approximations better than 2 for bounded weights.

In this paper, we provide the first fast *distributed* approximations for TAP. We present a distributed 2-approximation for weighted TAP which completes in $O(h)$ rounds, where h is the height of T. When h is large, we show a much faster 4-approximation algorithm for the unweighted case, completing in $O(D + \sqrt{n}\log^* n)$ rounds, where n is the number of vertices and D is the diameter of G.

Immediate consequences of our results are an $O(D)$-round 2-approximation algorithm for the minimum size 2-edge-connected spanning subgraph, which significantly improves upon the running time of previous approximation algorithms, and an $O(h_{MST} + \sqrt{n}\log^* n)$-round 3-approximation algorithm for the weighted case, where h_{MST} is the height of the MST of the graph. Additional applications are algorithms for verifying 2-edge-connectivity and for augmenting the connectivity of any connected spanning subgraph to 2.

Finally, we complement our study with proving lower bounds for distributed approximations of TAP.

1998 ACM Subject Classification G.2.2 Graph Theory: Graph algorithms

Keywords and phrases approximation algorithms, distributed network design, connectivity augmentation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.21

* Supported in part by the Israel Science Foundation (grant 1696/14).
† The full version of the paper is available at https://arxiv.org/abs/1711.03359.

1 Introduction

The tree augmentation problem (TAP) is a central problem in network design. In TAP, the input is a 2-edge-connected[1] graph G and a spanning tree T of G, and the goal is to augment T to be 2-edge-connected by adding to it a minimum size (or a minimum weight) set of edges from G. Augmenting the connectivity of T makes it resistant to any single link failure, which is crucial for network reliability. TAP is extensively studied in the sequential setting, with several classical 2-approximation algorithms [9, 13, 15, 18], as well as recent advances with the aim of achieving better approximation factors [1, 5, 8, 21].

TAP is part of a wider family of *connectivity augmentation* problems. Finding a minimum spanning tree (MST) is another prime example for a problem in this family, but, although an MST is a low-cost backbone of the graph, it cannot survive even one link failure. Hence, in order to guarantee stronger reliability, it is vital to find subgraphs with higher connectivity. The motivation for considering TAP is for the case that adding any new edge to the backbone incurs a cost, and hence if we are already given a subgraph with some connectivity guarantee then we would naturally like to augment it with additional edges of minimum number or weight, rather than to compute a well-connected low-cost subgraph from scratch. Connectivity augmentation problems also serve as building blocks in other connectivity problems, such as computing the minimum k-edge-connected subgraph. A natural approach is to start with building a subgraph that satisfies some connectivity guarantee (e.g., a spanning tree), and then augment it to have stronger connectivity.

Since the main motivation for TAP is improving the reliability of distributed networks, it is vital to consider TAP also from the distributed perspective. In this paper, we initiate the study of distributed connectivity augmentation and present the first distributed approximation algorithms for TAP. We do so in the CONGEST model [29], in which vertices exchange messages of $O(\log n)$ bits in synchronous rounds, where we show fast algorithms for both the unweighted and weighted variants of the problem. In addition to fast approximations for TAP, our algorithms have the crucial implication of providing efficient algorithms for approximating the minimum 2-edge-connected spanning subgraph, as well as several related problems, such as verifying 2-edge-connectivity and augmenting the connectivity of any spanning connected subgraph to 2. Finally, we complement our study with proving lower bounds for distributed approximations of TAP.

1.1 Our Contributions

Distributed approximation algorithms for TAP

Our first main contribution is the first distributed approximation algorithm for TAP. In particular, our algorithm provides a 2-approximation for weighted TAP in the CONGEST model, summarized as follows.

▶ **Theorem 1.1.** *There is a distributed 2-approximation algorithm for weighted TAP in the CONGEST model that runs in $O(h)$ rounds, where h is the height of the tree T.*

The approximation ratio of our algorithm matches the best approximation ratio for weighted TAP in the sequential setting. Its round complexity of $O(h)$ is tight if $h = O(D)$, where D is the diameter of G. This happens, for example, when T is a BFS tree, and follows from a lower bound of $\Omega(D)$ rounds which we show in Section 6.

[1] A graph G is 2-edge-connected if it remains connected after the removal of any single edge.

However, the height h of the spanning tree T may be large, even if the diameter of G is small, which raises the question of whether the dependence on h is necessary. We address this question by providing an algorithm for *unweighted* TAP that has a round complexity of $O(D + \sqrt{n}\log^* n)$ rounds, which is significantly smaller for large values of h. This only comes at the price of a slight increase in the approximation ratio, from 2 to 4.

▶ **Theorem 1.2.** *There is a distributed 4-approximation algorithm for unweighted TAP in the CONGEST model that runs in $O(D + \sqrt{n}\log^* n)$ rounds.*

Applications

The key application of our TAP approximation algorithm is an $O(D)$-round 2-approximation algorithm for the minimum size 2-edge-connected spanning subgraph problem (2-ECSS), which is obtained by building a BFS tree and augmenting it to a 2-edge-connected subgraph using our algorithm.

▶ **Theorem 1.3.** *There is a distributed 2-approximation algorithm for unweighted 2-ECSS in the CONGEST model that completes in $O(D)$ rounds.*

The time complexity of our algorithm improves significantly upon the time complexity of previous approximation algorithms for 2-ECSS, which are $O(n)$ rounds for a $\frac{3}{2}$-approximation [22] and $O(D + \sqrt{n}\log^* n)$ rounds for a 2-approximation [33].

In addition, our weighted TAP algorithm implies a 3-approximation for *weighted* 2-ECSS. Other applications of our algorithms are an $O(D)$-round algorithm for verifying 2-edge-connectivity, and an algorithm for augmenting the connectivity of any connected spanning subgraph H of G from 1 to 2.

Lower bounds

We complement our algorithms by presenting lower bounds for TAP. We first show that approximating TAP is a global problem which requires $\Omega(D)$ rounds even in the LOCAL model [25], where the size of messages is not bounded.

▶ **Theorem 1.4.** *Any distributed α-approximation algorithm for weighted TAP takes $\Omega(D)$ rounds in the LOCAL model, where $\alpha \geq 1$ can be any polynomial function of n. This holds also for unweighted TAP, if $1 \leq \alpha < \frac{n-1}{2c}$ for a constant $c > 1$.*

Theorem 1.4 implies that if $h = O(D)$ then our TAP approximation algorithms have an optimal round complexity. We also consider the case of $h = \omega(D)$ and show a family of graphs, based on the construction in [32], for which $\Omega(h)$ rounds are needed in order to approximate weighted TAP, were $h = \Theta(\frac{\sqrt{n}}{\log n})$.

▶ **Theorem 1.5.** *For any polynomial function $\alpha(n)$, there is a $\Theta(n)-$vertex graph of diameter $\Theta(\log n)$ for which any (perhaps randomized) distributed $\alpha(n)$-approximation algorithm for weighted TAP with an instance tree $T \subseteq G$ of height $h = \Theta(\frac{\sqrt{n}}{\log n})$ requires $\Omega(h)$ rounds in the CONGEST model.*

Theorem 1.5 implies that our algorithm for weighted TAP is optimal on these graphs. In particular, there cannot be an algorithm with a complexity of $O(f(h))$ for a sublinear function f.

1.2 Technical overview of our algorithms

As an introduction, we start by showing an $O(h)$-round 2-approximation algorithm for *unweighted* TAP, which allows us to present some of the key ingredients in our algorithms. Later, we explain how we build on these ideas and extend them to give an algorithm for the weighted case, and a faster algorithm for unweighted TAP.

Unweighted TAP

A natural approach for constructing a distributed algorithm for unweighted TAP could be to try to simulate the sequential 2-approximation algorithm of Khuller and Thurimella [18]. In their algorithm, the input graph G is first converted into a modified graph G'. Then, the algorithm finds a directed MST in G', which induces a corresponding augmentation in G.

When considered in the distributed setting, this approach imposes two difficulties. The first is that we cannot simply modify the input graph, because it is the graph that represents the underlying distributed network, whose topology is given and not under our control. The second is in the directed MST procedure, as finding a directed MST efficiently in the distributed setting seems to be difficult. The currently best known time complexity of this problem is $O(n^2)$ for an asynchronous setting [14], which is trivial in the CONGEST model.

We overcome the above using two key ingredients. First, we bring into our construction the tool of computing lowest common ancestors (LCAs). We show that building G' and simulating a distributed computation over it can be done by an efficient computation of LCAs, and we achieve the latter by leveraging the *labeling scheme* for LCAs presented in [2].

Second, we drastically diverge from the Khuller-Thurimella framework by replacing the expensive directed MST construction by a completely different procedure. Roughly speaking, we show that the simple structure of G' allows us to find an optimal augmentation in G' efficiently by scanning the input tree T from the leaves to the root and performing the following procedure. Each vertex sends to its parent information about edges that may be useful for the augmentation since they *cover* many edges of the tree, and the vertices use the LCA labels in order to decide which edges to add to the augmentation.

While a direct implementation of this would result in much information that is sent through the tree, we show that at most two edges need to actually be sent by each vertex. Thus, applying the labeling scheme and scanning the tree T result in a time complexity of $O(h)$ rounds, where h is the height of T. Finally, we prove that an optimal augmentation in G' gives a 2-approximation augmentation for G, which gives a 2-approximation for unweighted TAP in $O(h)$ rounds.

Weighted TAP

Our algorithm for the unweighted case relies heavily on the fact that we can compare edges and decide which one is the best for the augmentation according to the number of edges they cover in the tree. However, once the edges have weights, it is not clear how to compare edges. This is because of the tension between light edges that cover only few edges and heavier edges that cover many edges. Therefore, Theorem 1.1, which applies for the weighted case, cannot be directly obtained according to the above description. Hence, sending two edges per vertex up in the tree, as we do for the unweighted case, is insufficient. That is, the number of edges that should be considered for the augmentation could be much larger and vertices cannot decide locally which edges are useful without feedback from their ancestors in the tree.

Nevertheless, we show how to overcome this by introducing a technique of having each vertex send to its parent edges with *altered weights*. The trick here is that we modify the weight that is sent for an edge in a way that captures the cost for covering each edge of

the tree. This successfully addresses the competing needs of covering as many tree edges as possible, while using the lightest possible edges, and allows focusing on a smaller number of edges that may be useful for the augmentation. Finally, using standard pipelining, this gives a time complexity of $O(h)$ rounds for the weighted case as well.

Faster unweighted TAP

Both of our aforementioned algorithms rely on scanning the tree T, which results in a time complexity that is linear in the height h of the tree T. In order to avoid the dependence on h, one must be a able to add edges to the augmentation without scanning the whole tree.

However, if a vertex v does not get information about the edges added to the augmentation by the vertices in the whole subtree rooted at v, then it may add additional edges in order to cover tree edges that are already covered. But then we are no longer guaranteed to get an optimal augmentation in G', or even a good approximation for it.

Nevertheless, we are still able to show a faster algorithm for unweighted TAP, which completes in $O(D + \sqrt{n}\log^* n)$ rounds. The key ingredient in our algorithm is breaking the tree T into fragments and applying our 2-approximation for unweighted TAP algorithm on each fragment separately, as well as on the tree of fragments. Since our algorithm does not scan the whole tree, it may add different edges to cover the same tree edges, which makes the analysis much more involved. The approximation ratio analysis is based on dividing the edges to different types and bounding the number of edges of each type separately, using a subtle case-analysis. Although our algorithm does not find an optimal augmentation in G', it gives a 2-approximation for it, which results in a 4-approximation augmentation for the original graph G.

1.3 Related Work

Sequential algorithms for TAP

TAP is intensively studied in the sequential setting. Since TAP is NP-hard, approximation algorithms for it have been studied. The first 2-approximation algorithm for weighted TAP was given by Frederickson and JáJá [9], and was later simplified by Khuller and Thurimella [18]. Other 2-approximation algorithms for weighted TAP are the primal-dual algorithm of Goemans et al. [13], and the iterative rounding algorithm of Jain [15].

Recently, a new algorithm achieved a 1.5-approximation for unweighted TAP [21], and recent breakthroughs give approximations better than 2 for bounded weights [1, 8]. Achieving approximation better than 2 for the general weighted case is a central open question. See [17, 20] for surveys about approximation algorithms for connectivity problems. Also, the related work in [1] gives an overview of many recent sequential algorithms for TAP.

Related work in the distributed setting

While ours are the first distributed approximation algorithms for TAP itself, there are important related studies in the distributed setting.

MST. In the distributed setting, finding an MST, which is a minimum weight subgraph with connectivity 1, is a fundamental and well studied problem (see, e.g., [6, 7, 10, 11, 23, 28]). The first distributed algorithm for this problem is the GHS algorithm that works in $O(n \log n)$ time [10]. Following algorithms improved the round complexity to $O(D + \sqrt{n}\log^* n)$ [11, 23].

***k*-ECSS.** For the minimum weight 2-edge-connected spanning subgraph (2-ECSS) problem, there is a distributed algorithm of Krumke et al. [22]. Their approach is finding a specific spanning tree and then augmenting it to a 2-edge-connected graph. In the unweighted case, they augment a DFS tree following the sequential algorithm of Khuller and Vishkin [19], which results in an $O(n)$-round $\frac{3}{2}$-approximation algorithm for 2-ECSS. In the weighted case they augment an MST and suggest a general $O(n \log n)$-round 2-approximation algorithm for weighted TAP, which gives an $O(n \log n)$-round 3-approximation algorithm for 2-ECSS. Our algorithms for TAP imply faster approximations for unweighted and weighted 2-ECSS.

Another distributed algorithm for *unweighted* k-ECSS is an $O(k(D + \sqrt{n} \log^* n))$-round algorithm of Thurimella [33] that finds a sparse k-edge-connected subgraph. The general framework of the algorithm is to repeatedly find maximal spanning forests in the graph and remove their edges from the graph (this framework is also described in sequential algorithms [17, 26]). This gives a k-edge-connected spanning subgraph with at most $k(n-1)$ edges. Since any k-edge-connected subgraph has at least $\frac{kn}{2}$ edges, since the degree of each vertex is at least k, this approach guarantees a 2-approximation for unweighted k-ECSS.

Fault-tolerant tree structures. Another related problem is the construction of fault-tolerant tree structures. Distributed algorithms for constructing fault tolerant BFS and MST structures are given in [12], producing sparse subgraphs of the input graph G that contain a BFS (or an MST) of $G \setminus \{e\}$ for each edge e, for the purpose of maintaining the functionality of a BFS (or an MST) even when an edge fails. However, TAP is different from these problems in several aspects. First, we augment a specific spanning tree T rather then build the whole structure from scratch. In addition, since we need to preserve only connectivity when an edge fails and not the functionality of a BFS or an MST, optimal solutions for TAP may be much cheaper.

Additional related problems. Another connectivity augmentation problem studied in the distributed setting is the Steiner Forest problem [16, 24]. There are also distributed algorithms for finding the 2-edge-connected and 3-edge-connected components of a connected graph [30, 31], and distributed algorithms that decompose a graph with large connectivity into many disjoint trees, while almost preserving the total connectivity through the trees [4].

1.4 Preliminaries

For completeness, we first formally define the notion of edge connectivity.

▶ **Definition 1.6.** An undirected graph G is *k-edge-connected* if it remains connected after the removal of any $k-1$ edges.

The Tree Augmentation Problem (TAP). In TAP, the input is an undirected 2-edge-connected graph G with n vertices, and a spanning tree T of G. The goal is to add to T a minimum size (or a minimum weight) set of edges Aug from G, such that $T \cup Aug$ is 2-edge-connected. In the weighted version, each edge has a non-negative weight, and we assume that the weights of the edges can be represented in $O(\log n)$ bits.

▶ **Definition 1.7.** An edge e in a connected graph G is a *bridge* in G if $G \setminus \{e\}$ is disconnected.

▶ **Definition 1.8.** A non-tree edge $e = \{u, v\}$ *covers* the tree edge e' if e' is on the unique path in T between u and v, i.e., if e' is not a bridge in $T \cup \{e\}$.

A graph G is 2-edge-connected if and only if it does not contain bridges. Hence, augmenting the connectivity of T requires covering of all the tree edges.

Models of distributed computation. In the distributed CONGEST model [29], the network is modeled as an undirected connected graph $G = (V, E)$. Communication takes place in synchronous rounds. In each round, each vertex can send a message of $O(\log n)$ bits to each of its neighbors. The time complexity of an algorithm is measured by the number of rounds. Our algorithms work in the CONGEST model, where some of our lower bounds hold also in the stronger LOCAL model [25], where the size of messages is not bounded.

In the distributed setting, the input to TAP is a rooted spanning tree T of G with root r, whose height is denoted by h. The tree T is given to the vertices locally, that is, each vertex knows which of its adjacent edges is in T and which of those leads to its parent in T.[2] For each vertex $v \neq r$, we denote by $p(v)$ the parent of v in T. The output is a set of edges Aug, such that $T \cup Aug$ is 2-edge-connected. In the distributed setting it is enough that at the end of the algorithm each vertex knows which of the edges incident to it are added to Aug.

All the messages sent in our algorithms consist of a constant number of ids, labels and weights, hence the maximal message size is bounded by $O(\log n)$ bits, as required in the CONGEST model.

Roadmap. In Section 2, we describe our $O(h)$-round 2-approximation algorithm for unweighted TAP, and in Section 3 we extend it to the weighted case. In Section 4, we show applications of these algorithms, in particular for approximating 2-ECSS, and in Section 5 we present our faster algorithm for unweighted TAP. We present lower bounds for TAP in Section 6, and discuss questions for future research in Section 7. Full details of our algorithm for weighted TAP, as well as main proofs for it appear in Appendix A. Full details and proofs appear in the full version [3].

2 A 2-approximation for Unweighted TAP in $O(h)$ rounds

As an introduction, we start by describing an $O(h)$-round 2-approximation algorithm, A_{TAP}, for unweighted TAP. Here we give a high-level description of A_{TAP}. Full details and proofs appear in [3]. The general structure of A_{TAP} is as follows. It starts by building a related virtual graph G' as in [18]. Afterwards, we diverge completely from the approach of [18] since we cannot simulate it efficiently in the distributed setting, as explained in the introduction. Instead, A_{TAP} finds an optimal augmentation in G', and converts it to an augmentation Aug in G. We show that the size of Aug is at most twice the size of an optimal augmentation in G. All the communication in the algorithm is on the edges of the graph G, since G' is a virtual graph. In order to simulate the algorithm on G we use labels that represent the edges of G'.

Building G' from G. A_{TAP} starts by building a related *undirected* virtual graph G'. To simplify the presentation of the algorithm it is convenient to give an orientation to the edges of G'. However, we emphasize that G' is an undirected graph, that is, we do not address the notion of directed connectivity. The graph G' is defined as follows (as in [18]). The graph G' includes all the edges of T, and they are all oriented towards the root r of T. For every

[2] If a root and orientation are not given, we can find a root r and orient all the edges towards r in $O(h)$ rounds using standard techniques.

Figure 1 There are two cases for every non-tree edge in G. The left graph shows the first case, where the edge $\{u, v\}$ is between an ancestor and a descendant in T. The right graph shows the second case, where $t = LCA(u, v)$.

non-tree edge $e = \{u, v\}$ in G there are two cases (see Figure 1). If u is an ancestor of v in T, we add the edge $\{u, v\}$ to G', oriented from u to v. Otherwise, denote $t = LCA(u, v)$. In this case we add to G' the edges $\{t, u\}$ and $\{t, v\}$, oriented from t to u and to v, respectively.

In order to build the graph G' in the distributed setting, we use the *labeling scheme* for LCAs of Alstrup et al. [2]. This labeling scheme assigns labels of size $O(\log n)$ bits to the vertices of a rooted tree with n vertices, such that given the labels of u and v it is possible to infer the label of their LCA. The algorithm for computing the labels can be implemented in $O(h)$ rounds in the distributed setting.

Since G' is a virtual graph, the rest of the communication in the algorithm is only over the tree edges. In order to simulate the rest of the algorithm over G', it is enough that each vertex knows only the tree edges incident to it (which is its input), and the labels of the non-tree edges incoming to it in G'. Using the labeling scheme, each vertex learns this information by exchanging labels with all of its neighbors. For more details, see [3].

The Correspondence between G and G'. A non-tree edge $e = \{u, v\}$ in G covers all the edges in the unique path in T between u in v. To build G' from G, for each non-tree edge $e \in G$, we added one or two corresponding edges to G', which together cover exactly the same tree edges as e. This allows us to show that an optimal augmentation in G' gives a 2-approximation augmentation in G, when we replace each edge of the augmentation in G' by a corresponding edge in G. For full details and proofs see [3].

Finding an optimal augmentation in G'. In the graph G', all the edges that are not tree edges are between an ancestor and a descendant of it in T. This allows us to compare edges and define the notion of *maximal* edges. Intuitively, the notion of maximal edges would capture our goal that during the algorithm, when we cover a tree edge, we would like to cover it by an edge that reaches the highest ancestor possible, allowing us to cover many tree edges simultaneously. This motivates the following definition. Let v be a vertex in the tree, and let $e = \{u, w\}$ and $e' = \{u', w'\}$ be two edges between ancestors u, u' of v and descendants w, w' of v. We say that e is the *maximal* edge among e and e' if and only if u is an ancestor of u'. If $u = u'$ we can choose arbitrarily one of them to be the maximal edge.

In order to cover all tree edges of G', we assign each vertex $v \neq r$ in G' with the responsibility of covering the tree edge $\{v, p(v)\}$. In our algorithm A_{Aug} for finding an optimal augmentation in G', the tree T is scanned from the leaves to the root, and whenever a tree edge that is still not covered is reached, it is covered by the vertex responsible for it, using the maximal edge possible. For doing so, it is enough that each vertex v sends to its parent information about at most two edges. One of them is the maximal edge that was already added to the augmentation in order to cover the tree edge $\{v, p(v)\}$, and one is the maximal edge possible to cover the tree edge $\{v, p(v)\}$. We show that using the LCA labels of edges, vertices can compute easily if a tree edge is covered, and which edge is the maximal edge. The full description of the algorithm A_{Aug} is given in [3].

Correctness proof. Denote by A the solution obtained by A_{Aug}, and by A^* an optimal augmentation in G'. We show in [3] that A is an optimal augmentation in G'. The key ingredient is to show a one-to-one mapping from A to A^*. For each edge $e \in A$, we look at the tree edge $t(e) = \{v, p(v)\}$ where v is the vertex that decides to add e to the augmentation. We map e to an edge $e^* \in A^*$ that covers $t(e)$. Showing that this mapping is one-to-one is based on the fact that when we add a new edge to the augmentation, it is the maximal edge possible. In addition, we show that the time complexity of the algorithm is $O(h)$ rounds. Since an optimal augmentation in G' corresponds to a 2-approximation augmentation in G, this gives the following.

▶ **Theorem 2.1.** *There is a distributed 2-approximation algorithm for unweighted TAP in the CONGEST model that runs in $O(h)$ rounds, where h is the height of the tree T.*

3 A 2-approximation for Weighted TAP in $O(h)$ rounds

In this section, we give a high-level description of our algorithm for weighted TAP. Full details and main proofs appear in Appendix A. Our algorithm for weighted TAP, A_{wTAP}, has the same structure of A_{TAP}. It starts by building the same virtual graph G', and then it finds an optimal augmentation in G'. The only difference in building G' is that now each edge e is replaced by one or two edges in G' with the same weight that e has. The proof that an optimal augmentation in G' corresponds to an augmentation in G with at most twice the cost of an optimal augmentation in G is the same as in the unweighted case.

The difference is in finding an optimal augmentation in G'. In the unweighted case, for each vertex v, the only edge incoming to v in G' that was useful for the algorithm was the maximal edge. However, when edges have weights, potentially all the edges incoming to v may be useful for the algorithm, and we can no longer use the notion of *maximal* edges in order to compare edges. This is because of the tension between heavy edges that cover many edges of the tree, and light edges that cover less edges of the tree. To overcome this obstacle, we introduce a new technique of *altering* the weights of the edges we send in the algorithm. We next describe how our new approach allows us to find an optimal augmentation in G'.

Finding an Optimal Augmentation in G'. In the weighted case, it is not clear anymore how to compare edges. A vertex v may have many optional edges that cover $\{v, p(v)\}$, where one of them has the minimum weight, but other edges cover more tree edges. In order to cover the tree edge between v and its parent, it is best to add the edge with minimum weight. However, in order to cover additional tree edges, we may want to add one of the other edges. Since v alone does not have enough information to resolve this trade-off, one could have the vertices propagate all the edges up in the tree, but this would accumulate to too much information, which would render the algorithm very slow.

Instead, we pinpoint the exact source of the trade-off between length and weight of covering edges, as follows. Let min_v be the weight of the minimum weight edge that covers $\{v, p(v)\}$. The intuition behind our approach is that in order to cover the tree edge $\{v, p(v)\}$ we must pay at least min_v. Thus, min_v captures the cost of covering this tree edge. Therefore, before sending to its parent information about relevant edges, v alters their weights by reducing from them the weight min_v. In terms of $p(v)$, the reduced weights represent the extra cost incurred when choosing any of these edges in order to cover additional tree edges. Using the reduced weights in our algorithm instead of the original ones is crucial for selecting which edges to add to the augmentation, and allows to divide the weight of an edge in a way that captures the cost for covering each tree edge. In addition, we show that using this approach,

sending information about at most h edges from each vertex to its parent suffices for selecting the best edges for the augmentation.

The Algorithm. Our algorithm consists of two traversals of the tree: from the leaves to the root and vice versa. As in A_{Aug}, each vertex v is responsible for covering the tree edge $\{v, p(v)\}$.

In the first traversal, each vertex v computes the weight min_v of the minimum weight edge that covers the tree edge $\{v, p(v)\}$ according to the weights of the edges it receives from its children, and the weights of the edges incoming to it. It also computes the weights of the minimum weight edges that cover the path from v to each of its ancestors u, according to the weights v receives in the algorithm. Then, v subtracts min_v from the weights of these edges, and sends them to its parent with the altered weights.

In the second traversal, we scan the tree from the root to the leaves. Each child v of r adds to the augmentation the edge having weight min_v. It informs the relevant child who sent it, if exists, and informs its other children it did not add their edges. Each internal vertex v receives from its parent a message that indicates whether one of the edges it sent was added to the augmentation by one of its ancestors or not. In the former case, v learns that this edge was added to the augmentation and forwards the message to the relevant child who sent it, if such exists. Otherwise, the tree edge $\{v, p(v)\}$ is still not covered, and v adds to the augmentation the edge having weight min_v. It informs the relevant child who sent it, if exists, and informs its other children that their edges were not added to the augmentation.

A full description of the algorithm is given in Appendix A.1.

Time analysis. In our algorithm, each vertex sends to its parent information about at most h edges. If each vertex waits to receive all the messages from its children, before sending messages to its parent, it would result in a time complexity of $O(h^2)$ rounds. However, using pipelining we get a time complexity of $O(h)$ rounds. The main intuition is that although each vertex v may receive h different messages from each of its children during the algorithm, in order for v to send to its parent $p(v)$ the message concerning an ancestor u, the vertex v only needs to receive one message from each of its children concerning the ancestor u. Hence, if all the vertices send the messages according to increasing order of heights of their ancestors, we can pipeline the messages and get a time complexity of $O(h)$ rounds. The full proof appears in [3].

Correctness proof. In Appendix A.3, we give a correctness proof for the algorithm. The challenge in establishing the correctness of our algorithm lies in the fact that the vertices use altered weights rather than the original ones. Nevertheless, we show that our intuition behind choosing these altered weights faithfully captures the essence of finding an augmentation in the weighted case. The key ingredient we use in our proof is giving a *cost* to each edge of T. For a tree edge $t = \{v, p(v)\}$, we assign the cost $c(t) = min_v$. We are then able to prove the following.

(I) The sum of the costs of the edges of T is equal to the cost of the augmentation obtained by the algorithm.

(II) The cost of any augmentation of G' is at least the sum of costs of the edges of T.

To show (1), for each edge e added to the augmentation, we assign a tree path P_e, as follows. For an edge $e = \{u, x\}$ added to the augmentation by a vertex v, where u is an ancestor of x in the tree, we assign the tree path P_e between x to $p(v)$. Then, we show that $w(e) = \sum_{t \in P_e} c(t)$. The proof is based on the fact that the cost of the tree edge $\{v, p(v)\}$

is min_v by definition, however, its cost is also $w(e) - \sum_{v' \in V'} min_{v'}$ where V' are all the vertices on the path between x and v, excluding v. This is since each vertex v' in this path reduces $min_{v'}$ from the weight of e before sending it to its parent, and since e is the edge v chooses to add to the augmentation. This gives $w(e) = min_v + \sum_{v' \in V'} min_{v'} = \sum_{t \in P_e} c(t)$. In addition, we show that the paths P_e are disjoint and their union is the entire tree T, which proves (1). The proof is based on the fact that the edges of the augmentation cover all tree edges, and that a vertex v adds an edge to the augmentation only if the tree edge $\{v, p(v)\}$ is not already covered by an edge added by one of its ancestors. We prove (2) in a similar way.

In conclusion, our algorithm gives an optimal augmentation in G', which gives a 2-approximation augmentation in G.

▶ **Theorem 1.1.** *There is a distributed 2-approximation algorithm for weighted TAP in the CONGEST model that runs in $O(h)$ rounds, where h is the height of the tree T.*

4 Applications

In this section, we discuss applications of our algorithms, and show they provide efficient algorithms for additional related problems.

Minimum Weight 2-Edge-Connected Spanning Subgraph. In the minimum weight 2-edge-connected spanning subgraph problem (2-ECSS), the input is a 2-edge-connected graph G, and the goal is to find the minimum weight 2-edge-connected spanning subgraph of G. Using A_{TAP} we have the following.

▶ **Theorem 1.3.** *There is a distributed 2-approximation algorithm for unweighted 2-ECSS in the CONGEST model that completes in $O(D)$ rounds.*

Proof. We apply A_{TAP} on G and a BFS tree T of G. Finding a BFS tree takes $O(D)$ rounds [29], and A_{TAP} takes $O(D)$ rounds since T is a BFS tree. The size of the augmentation Aug is at most $n-1$ because in the worst case we add a different edge in order to cover each tree edge. Hence, $T \cup Aug$ is a 2-edge-connected subgraph with at most $2(n-1)$ edges. Note that any 2-edge-connected graph has at least n edges, which implies a 2-approximation, as claimed. ◀

The above algorithm has a better time complexity compared to the algorithm of [22], which finds a $\frac{3}{2}$-approximation to 2-ECSS in $O(n)$ rounds. In the algorithm of [22], the augmented tree T is a DFS tree rather then a BFS tree. The same proof of [19, 22] gives that if we apply A_{TAP} on G and a DFS tree we also obtain a $\frac{3}{2}$-approximation to 2-ECSS in $O(n)$ rounds. For weighted 2-ECSS, using A_{wTAP} gives the following.

▶ **Theorem 4.1.** *There is a distributed 3-approximation algorithm for weighted 2-ECSS in the CONGEST model that completes in $O(h_{MST} + \sqrt{n}\log^* n)$ rounds, where h_{MST} is the height of the MST.*

Proof. We follow the same approach of [22]. We start by constructing an MST, which takes $O(D + \sqrt{n}\log^* n)$ rounds [23], and then we augment it using A_{wTAP} in $O(h_{MST})$ rounds.[3] Let $w(A)$ be the weight of an optimal solution A to weighted 2-ECSS. Since both the MST and an optimal augmentation have weights at most $w(A)$, and since our algorithm for

[3] We assume that the MST is unique. Otherwise, h_{MST} is the height of the MST we construct.

weighted TAP gives a 2-approximation, this approach gives a 3-approximation for weighted 2-ECSS. ◀

This algorithm has a better time complexity compared to the algorithm of [22], which takes $O(n \log n)$ rounds, with the same approximation ratio.

Increasing the Edge-Connectivity from 1 to 2. A_{wTAP} is a 2-approximation algorithm for TAP, but can also be used to increase the connectivity of any spanning subgraph H of G from 1 to 2. In order to do so, we start by finding a spanning tree T of H. Note that it is not enough to apply A_{TAP} on T and take the augmentation obtained, since edges from H can be added to the augmentation with no cost. Hence, we apply A_{wTAP} on G and T, where we set the weights of all the edges of H to be 0. The augmentation Aug we obtain is a set of edges such that $T \cup Aug$ is 2-edge-connected, which also implies that $H \cup Aug$ is 2-edge-connected. In addition, its cost is at most twice the cost of an optimal augmentation of H, because any augmentation of H corresponds to an augmentation of T with the same cost, and Aug is a 2-approximation to the optimal augmentation of T. The time complexity is $O(D_H)$ rounds where D_H is the diameter of H, since finding a spanning tree T of H takes $O(D_H)$ rounds and applying A_{wTAP} takes $O(D_H)$ rounds because it is the height of T.

Verifying 2-Edge-Connectivity. The algorithm A_{TAP} can be used in order to verify if a connected graph G is 2-edge-connected in $O(D)$ rounds, where at the end of the algorithm all the vertices know if G is 2-edge-connected.[4] We start by building a BFS tree T of G and then apply A_{TAP} to G and T. Note that when we find an optimal augmentation in G' by A_{Aug}, each vertex v is responsible to cover the tree edge $\{v, p(v)\}$. If the graph G is 2-edge-connected, all the edges can be covered. If the graph G is not 2-edge-connected, then there is a tree edge $\{v, p(v)\}$ that is a bridge in the graph, and hence cannot be covered by any edge in G. In such a case, v identifies that it cannot cover the edge and hence the graph is not 2-edge-connected. Therefore, after scanning the tree from the leaves to the root in A_{Aug}, we can distinguish between these two cases, which takes $O(D)$ rounds. The root r can distribute the information to all the vertices in $O(D)$ rounds as well.

5 A 4-approximation for Unweighted TAP in $\widetilde{O}(D + \sqrt{n})$ rounds

The time complexity of A_{TAP} and A_{wTAP} is linear in the height of T. When h is large, we suggest a much faster $O(D + \sqrt{n} \log^* n)$-round algorithm for unweighted TAP. Here we give a high-level description of the algorithm, full details and proofs appear in [3].

The structure of the algorithm is the same as the structure of A_{TAP}. It starts by building the same virtual graph G', and then it finds an augmentation in G'. However, since we want to reduce the time complexity, our algorithm cannot scan the whole tree anymore. Therefore, we can no longer use directly the LCA labeling scheme and the algorithm A_{Aug} for finding an optimal augmentation. To overcome this, we break the tree T into fragments, and we divide the algorithm into local parts, in which we communicate in each fragment separately, and to global parts, in which we coordinate between different fragments over a BFS tree. This approach is useful also in other distributed algorithms for global problems, such as finding an MST [23] or a minimum cut [27]. The challenge is showing that this approach guarantees a

[4] A verification algorithm with the same complexity can also be deduced from the edge-biconnectivity algorithm of Pritchard [30].

good approximation. Since our algorithm does not scan the whole tree T it may add different edges in order to cover the same tree edges, which makes the analysis much more involved.

Building G' from G. In order to build G' from G we use the labeling scheme for LCAs that we used in A_{TAP}. However, applying this scheme directly takes $O(h)$ rounds. We show how to compute all the relevant LCAs more efficiently in $O(D + \sqrt{n})$ rounds. The idea is to apply the labeling scheme on each fragment separately to obtain *local labels*, and to apply the labeling scheme on the tree of fragments to obtain *global labels*. We show that using the local and global labels, and additional information on the structure of the tree of fragments, each vertex can compute all the edges incoming to it in G'. For full details see [3].

Finding an optimal augmentation in G'. In order to find an augmentation in G', we need to cover tree edges between fragments (*global edges*) and tree edges in the same fragment (*local edges*). For doing so, we start by computing all the maximal edges that cover the global edges. To cover all the global edges, one approach could be to add all these maximal edges to the augmentation. However, this cannot guarantee a good approximation. Instead, we apply A_{Aug} on the tree of fragments in order to cover all the global edges. Then, we apply it on each fragment separately in order to cover the local edges in the fragment that are still not covered. This algorithm requires coordination between different fragments, since each vertex v needs to learn if the tree edge $\{v, p(v)\}$ is already covered after the first part of the algorithm. In addition, although the second part is applied on each fragment separately, a vertex v may need to add an edge incoming to another fragment to cover the tree edge $\{v, p(v)\}$. For achieving an efficient time complexity, we show how to use only $O(\sqrt{n})$ different messages for the whole coordination of the algorithm. For full details see [3].

Approximation Ratio. We show that this approach gives a 2-approximation to the optimal augmentation in G'. Denote by A the solution obtained by the algorithm and by A^* an optimal augmentation in G'. In the correctness proof of A_{TAP} we show a one-to-one mapping from A to A^*, but this mapping is no longer one to one here. However, if we could show that each edge in A^* is mapped to by at most two edges from A, we can obtain a 2-approximation. Unfortunately, this does not hold either.

Our approach is to divide the edges in A to two types A_1, A_2 as follows. We map each edge $e \in A$ to a corresponding path P_e in T. If P_e contains an internal vertex with more than one child in the tree we say that $e \in A_1$, otherwise $e \in A_2$. Then, we show that $|A_1| \leq 2|A^*|$, and $|A_2| \leq 2|A^*|$. The main idea is that the number of edges in A_1 is related to the degrees of internal vertices in T, which affects the number of leaves in the tree. We use this in order to show that $|A_1| \leq 2\ell$ where ℓ is the number of leaves in T. Note that ℓ is a lower bound on the size of any augmentation in G', since we need to add to the augmentation a different edge in order to cover each one of the leaves. This gives $|A_1| \leq 2|A^*|$.

In order to show that $|A_2| \leq 2|A^*|$, we use the fact that the edges of A_2 correspond to tree paths with a simple structure. This allows us to show a mapping between A_2 to A^* in which each edge in A^* is mapped to by at most two edges from A_2, giving $|A_2| \leq 2|A^*|$.

In conclusion, $|A| = |A_1 \cup A_2| \leq 4|A^*|$. A more delicate analysis extending these ideas gives $|A| \leq 2|A^*|$. This gives a 2-approximation to the optimal augmentation of G', which results in a 4-approximation to the optimal augmentation in G. A full proof of the approximation ratio is given in [3], and gives the following.

▶ **Theorem 1.2.** *There is a distributed 4-approximation algorithm for unweighted TAP in the CONGEST model that runs in $O(D + \sqrt{n}\log^* n)$ rounds.*

6 Lower Bounds

An $\Omega(D)$ Lower Bound for TAP in the LOCAL model. We show that TAP is a global problem, which admits a lower bound of $\Omega(D)$ rounds, even in the LOCAL model where the size of messages is unbounded.

▶ **Theorem 1.4.** *Any distributed α-approximation algorithm for weighted TAP takes $\Omega(D)$ rounds in the LOCAL model, where $\alpha \geq 1$ can be any polynomial function of n. This holds also for unweighted TAP, if $1 \leq \alpha < \frac{n-1}{2c}$ for a constant $c > 1$.*

The proof is based on considering two similar graphs G_1, G_2, defined as follows. The graph G_1 consists of a path T of $n = 2k+1$ vertices $\{v_0, v_1, ..., v_{2k}\}$, and the edges $\{v_{2i}, v_{2(i+1)}\}$ for $0 \leq i < k$, where $G_2 = G_1 \cup \{v_0, v_{2k}\}$. In G_2, adding the edge $\{v_0, v_{2k}\}$ already covers all the edges of the tree T, where in G_1 we must add many edges to the augmentation. However, a vertex in the middle of the path T cannot distinguish between the graphs G_1, G_2 in less than $\Omega(D)$ rounds. Based on these ideas, we show that approximating TAP takes $\Omega(D)$ rounds. The full proof appears in [3].

A Lower Bound for weighted TAP in the CONGEST model. By Theorem 1.4, when $h = O(D)$ our algorithms A_{TAP}, A_{wTAP} are optimal up to a constant factor. But what about the case of $h = \omega(D)$ for the CONGEST model? In [3], we show a family of graphs where $h = \omega(D)$, in which $\Omega(h)$ rounds are needed in order to approximate weighted TAP, where $h = O(\sqrt{n})$. The lower bound is proven using a reduction from the 2-party set-disjointness problem, in which there are two players, Alice and Bob, that have to decide whether their input strings are disjoint. Our construction is based on a construction presented in [6, 32]. In order to use this construction for showing lower bounds for TAP, we add to it additional parallel edges[5] and give weights to the edges in such a way that all the edges of the input tree T can be covered by parallel edges of weight 0, except for k edges, $\{e_i\}_{i=1}^{k}$. The edge e_i may be covered either by a corresponding parallel edge e_i^A, or by a distant edge e_i^B that closes a cycle that contains e_i (see Figure 2 for an illustration). However, the weights of the edges e_i^A and e_i^B depend on the i'th bit in the input strings of Alice and Bob, such that there is a light edge that covers e_i if and only if this bit equals 0 at least in one of the input strings. It follows that all the k edges can be covered by light edges if and only if the input strings of Alice and Bob are disjoint.

Using a known lower bound on the communication complexity of set-disjointness, we prove Theorem 1.5. Full details and proofs appear in [3].

▶ **Theorem 1.5.** *For any polynomial function $\alpha(n)$, there is a $\Theta(n)-$vertex graph of diameter $\Theta(\log n)$ for which any (perhaps randomized) distributed $\alpha(n)$-approximation algorithm for weighted TAP with an instance tree $T \subseteq G$ of height $h = \Theta(\frac{\sqrt{n}}{\log n})$ requires $\Omega(h)$ rounds in the CONGEST model.*

7 Discussion

In this paper, we present the first distributed approximation algorithms for TAP. Many intriguing problems remain open. First, can we get efficient distributed algorithms for TAP with an approximation ratio better than 2? In the sequential setting, achieving an approximation better than 2 for weighted TAP is a central open question. However, there

[5] We also show a construction with no parallel edges.

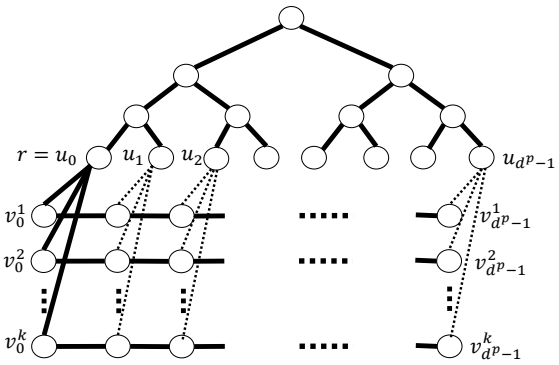


Figure 2 The structure of the graph G. The edges of the input tree T are marked with solid lines, other edges are marked with dashed lines. The edge $e_i = \{r, v_0^i\}$ can be covered either by a parallel edge e_i^A, or by the corresponding edge $e_i^B = \{u_{d^p-1}, v_{d^p-1}^i\}$ (additional options are expensive).

are several recent algorithms achieving better approximations for unweighted TAP [5, 21] or for weighted TAP with bounded weights [1, 8].

Second, can we show algorithms for weighted TAP and weighted 2-ECSS that are more efficient? For *unweighted* TAP, we showed an algorithm with a time complexity of $O(D + \sqrt{n}\log^* n)$ rounds. In addition, for *unweighted* k-ECSS, the $O(k(D + \sqrt{n}\log^* n))$-round algorithm of Thurimella [33] gives a 2-approximation. However, currently there are no algorithms for the *weighted* problems with a similar time complexity.

There are many additional connectivity augmentation problems, such as increasing the edge connectivity from k to $k + 1$ or to some function $f(k)$, as well as augmentation for increasing the vertex connectivity. Such problems have been widely studied in the sequential setting, and a natural question is to design distributed algorithms for them.

Finally, it is interesting to study TAP and additional connectivity problems also in other distributed models, such as the dynamic model where edges or vertices may be added or removed from the network during the algorithm. An interesting question is how to maintain highly-connected backbones when the network can change dynamically.

References

1 David Adjiashvili. Beating approximation factor two for weighted tree augmentation with bounded costs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2384–2399. SIAM, 2017.

2 Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.

3 Keren Censor-Hillel and Michal Dory. Fast distributed approximation for TAP and 2-edge-connectivity. *arXiv:1711.03359*, 2017. URL: `https://arxiv.org/abs/1711.03359`.

4 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing (PODC)*, pages 156–165. ACM, 2014.

5 Joseph Cheriyan and Zhihan Gao. Approximating (unweighted) tree augmentation via lift-and-project, part II. *Algorithmica*, pages 1–44, 2015.

6 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.

7 Michael Elkin. A simple deterministic distributed MST algorithm, with near-optimal time and message complexities. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 157–163. ACM, 2017. doi:10.1145/3087801.3087823.

8 Samuel Fiorini, Martin Groß, Jochen Könemann, and Laura Sanità. A 3/2-approximation algorithm for tree augmentation via chvátal-gomory cuts. *CoRR*, abs/1702.05567, 2017. arXiv:1702.05567.

9 Greg N Frederickson and Joseph JáJá. Approximation algorithms for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.

10 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.

11 Juan A Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

12 Mohsen Ghaffari and Merav Parter. Near-optimal distributed algorithms for fault-tolerant tree structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 387–396. ACM, 2016.

13 Michel X Goemans, Andrew V Goldberg, Serge A Plotkin, David B Shmoys, Eva Tardos, and David P Williamson. Improved approximation algorithms for network design problems. In *SODA*, volume 94, pages 223–232, 1994.

14 P Humblet. A distributed algorithm for minimum weight directed spanning trees. *IEEE Transactions on Communications*, 31(6):756–762, 1983.

15 Kamal Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Combinatorica*, 21(1):39–60, 2001.

16 Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distributed Computing*, 25(3):189–205, 2012.

17 Samir Khuller. Approximation algorithms for finding highly connected subgraphs. In *Approximation algorithms for NP-hard problems*, pages 236–265. PWS Publishing Co., 1996.

18 Samir Khuller and Ramakrishna Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.

19 Samir Khuller and Uzi Vishkin. Biconnectivity approximations and graph carvings. *Journal of the ACM (JACM)*, 41(2):214–235, 1994.

20 Guy Kortsarz and Zeev Nutov. Approximating minimum cost connectivity problems. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.

21 Guy Kortsarz and Zeev Nutov. A simplified 1.5-approximation algorithm for augmenting edge-connectivity of a graph from 1 to 2. *ACM Transactions on Algorithms (TALG)*, 12(2):23, 2016.

22 Sven O Krumke, Peter Merz, Tim Nonner, and Katharina Rupp. Distributed approximation algorithms for finding 2-edge-connected subgraphs. In *International Conference On Principles Of Distributed Systems (OPODIS)*, pages 159–173. Springer, 2007.

23 Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC)*, pages 238–251. ACM, 1995.

24 Christoph Lenzen and Boaz Patt-Shamir. Improved distributed steiner forest construction. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing (PODC)*, pages 262–271. ACM, 2014.

25 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

26 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(1):583–596, 1992.

27 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014.

28 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time-and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 743–756. ACM, 2017.

29 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

30 David Pritchard. Robust network computation. Master's thesis, MIT, 2005.

31 David Pritchard and Ramakrishna Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transactions on Algorithms (TALG)*, 7(4):46, 2011.

32 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.

33 Ramakrishna Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC)*, pages 28–37. ACM, 1995.

A A 2-approximation for weighted TAP in $O(h)$ rounds: full details

In this section, we give full details and main proofs for the algorithm A_{wTAP}. We show the following.

▶ **Theorem 1.1.** *There is a distributed 2-approximation algorithm for weighted TAP in the CONGEST model that runs in $O(h)$ rounds, where h is the height of the tree T.*

As explained in Section 3, the algorithm A_{wTAP} has the same structure of A_{TAP}, but it differs from it in the algorithm for finding an optimal augmentation in G'. In Section A.1, we describe our algorithm for finding an optimal augmentation in G'. In Section A.2, we analyze the time complexity of the algorithm, and in Section A.3 we prove its correctness.

A.1 The algorithm

A description of the algorithm is given in Algorithm 1. For simplicity of presentation, we start by describing an algorithm which takes $O(h^2)$ rounds. Later, in Section A.2, we explain how using pipelining we improve the time complexity to $O(h)$ rounds.

Technical Details:

We assume in the algorithm that each vertex knows all the ids of its ancestors in T. We justify it in the next claim. Note that when we construct G', if $\{u, v\}$ is an edge between an ancestor u and its descendant v in T, v learns the label of u according to the LCA labeling scheme and not the id of u. However, once v learns about the ids and labels of all its ancestors, it knows the id of u as well, and can use it in the algorithm.

▶ **Claim 1.1.** *All the vertices can learn the ids and labels of all their ancestors in $O(h)$ rounds.*

Algorithm 1 Finding an Optimal Augmentation in G'.

The code is for every vertex $v \neq r$

1: Initialization:
2: $e_{v,u} \leftarrow$ the minimum weight edge incoming to v that covers the path between v and its ancestor u or $\perp$ if there is no such edge.
3: $w_v(u) \leftarrow w(e_{v,u})$ for each ancestor u of v such that $e_{v,u} \neq \perp$, and $w_v(u) \leftarrow \infty$ otherwise.
4: $A_v \leftarrow$ the union of v and its children in T.
5: $Aug_v \leftarrow \emptyset$

6: First Traversal:
7: **if** v is a leaf **then**
8: for each ancestor u of v: $sender_v(u) \leftarrow v$
9: **else**
10: **wait** for receiving $w_{v'}(u)$ for all ancestors u of v, from each child v' of v
11: for each ancestor u of v: $w_v(u) \leftarrow min_{v' \in A_v} w_{v'}(u)$, $sender_v(u) \leftarrow argmin_{v' \in A_v} w_{v'}(u)$
12: **end if**
13: $min_v \leftarrow w_v(p(v))$
14: for each ancestor u of v: $w_v(u) \leftarrow w_v(u) - min_v$
15: for each ancestor $u \neq p(v)$ of v **send** $(u, w_v(u))$ to $p(v)$

16: Second Traversal:
17: $u \leftarrow p(v)$
18: **if** v is not a child of r **then**
19: **wait** for a message m from $p(v)$
20: **if** $m \neq \perp$ **then** $u \leftarrow m$
21: **end if**
22: **end if**
23: $s \leftarrow sender_v(u)$
24: **if** $s = v$ **then**
25: $Aug_v \leftarrow Aug_v \cup \{e_{v,u}\}$
26: **else**
27: send u to s
28: **end if**
29: for each child $v' \neq s$ of v **send** $\perp$ to v'

Proof. In order to do this, at the first round each vertex sends to its children its id and label. In the next round, each vertex sends to its children the id and label it received in the previous round, and we continue in the same way until each vertex learns about all its ancestors. Clearly, after h rounds each vertex learns all the ids and labels of all its ancestors. ◀

▶ **Claim 1.2.** *If a vertex v adds $e_{v,u}$ to Aug_v in line 25 of its algorithm, then $e_{v,u} \neq \perp$.*

Proof. Since G' is 2-edge-connected, we can cover all tree edges by edges from G'. Hence, the minimum weight of an edge that covers some tree edge is never infinite. It follows that if a vertex v adds $e_{v,u}$ to Aug_v, then $e_{v,u} \neq \perp$. ◀

A.2 Time analysis

We next analyze the time complexity of the algorithm. In the second traversal of the tree, each parent sends to each of its children one message, which takes $O(h)$ rounds. In the first traversal of the tree, each vertex sends to its parent at most h edges. If each vertex waited until it got all the messages from its children before sending messages to its parent, it would result in a time complexity of $O(h^2)$ rounds. We show how to get a time complexity of $O(h)$ rounds for this part using pipelining. To show this, we carefully design each vertex v to send the messages $(u, w_v(u))$ in increasing order of heights of its ancestors.

The main intuition is that although each vertex v may receive h different messages from each of its children during the algorithm, in order for v to send to its parent $p(v)$ the message concerning an ancestor u, the vertex v only needs to receive one message from each of its children concerning the ancestor u. Hence, if all the vertices send the messages according to increasing order of heights of their ancestors, we can pipeline the messages and get a time complexity of $O(h)$ rounds. We formalize this intuition in the next lemma, which we prove in the full version.

▶ **Lemma A.3.** *A vertex v at height i sends to its parent until round $i + j$ the message $(u, w_v(u))$ such that u is an ancestor of v at height j.*

From the lemma we get that by round $2h$ all the children of r learn about the minimum weight edge that covers the tree edge between them and r, so the first traversal is completed after $O(h)$ rounds. It follows that the overall time complexity of the algorithm is $O(h)$ rounds as needed, giving the following.

▶ **Lemma A.4.** *Algorithm 1 completes in $O(h)$ rounds.*

A.3 Correctness Proof

▶ **Lemma A.5.** *Algorithm 1 finds an optimal augmentation in G'.*

Proof. Note that the solution obtained by the algorithm is an augmentation of G' because each vertex v adds an edge in order to cover the tree edge $\{v, p(v)\}$ if it is not already covered by an edge which one of its ancestors decides to add to the augmentation. In order to show that the augmentation is optimal we give costs to the edges of T such that the sum of the costs is equal to both the cost of the solution obtained by the algorithm and the cost of an optimal augmentation of G'. Hence, we conclude that the cost of the solution obtained by the algorithm is optimal.

Giving costs to the edges of T:

Fix a vertex $v \neq r$ and let $t = \{v, p(v)\}$. We define $c(t) = min_v$ (the value of $w_v(p(v))$ in line 13 of the algorithm).

For an edge $e = \{u, x\}$ that covers t, such that u is an ancestor of x in T, let P be the path of tree edges between x and $p(v)$ in T. For a vertex v' such that $\{v', p(v')\} \in P$, let $P_{v'}$ be the path of tree edges between x and v'. Note that min_v is the weight of the minimum weight edge covering the tree edge $t = \{v, p(v)\}$ according to the weights v receives in the algorithm. Denote this edge by e_v.

▶ **Claim 1.6.** $w(e_v) = \sum_{t' \in P} c(t')$.

Proof. Let $e_v = \{u, x\}$, where u is an ancestor of x in T. For each vertex v' on the path between x and v, e_v is the minimum weight edge covering the path between v' and its ancestor $p(v)$, according to the weights v' receives in the algorithm, as otherwise we get a contradiction to the definition of e_v. Each vertex on this path reduces $min_{v'}$ from the weight of e_v it receives before sending it to its parent. Denote by V' all the vertices on the path between x and v, excluding v. It follows that

$$c(t) = min_v = w(e_v) - \sum_{v' \in V'} min_{v'} = w(e_v) - \sum_{t' \in P_v} c(t'),$$

which gives $w(e_v) = \sum_{t' \in P} c(t')$. ◀

▶ **Claim 1.7.** *The sum of the costs of the edges of T is equal to the cost of the solution obtained by the algorithm.*

Proof. We map each edge e added to the augmentation to a path P_e of tree edges, such that:

(I) The paths that correspond to different augmentation edges are disjoint, and their union is the entire tree T. That is, $P_e \cap P_{e'} = \emptyset$ for $e \neq e'$, and $\cup P_e = T$.

(II) The weight of e is equal to the sum of costs of tree edges in the corresponding path, i.e., $w(e) = \sum_{t' \in P_e} c(t')$.

Let $e = \{u, x\}$ be an edge added to the augmentation, such that u is an ancestor of x in T. Let v be the vertex that decides to add e to the augmentation. Note that v decides to add e to the augmentation because it covers the tree edge $\{v, p(v)\}$, which is not covered yet by an edge that one of v's ancestors decides to add to the augmentation. We map e to the tree path P_e that consists of all the tree edges on the path between x and $p(v)$. Note that e covers all the edges on this path (and it may also cover other tree edges, on the path between $p(v)$ and u in T). This divides the tree edges to disjoint paths because the vertices on the path between x and $p(v)$ do not decide to add other edges to the augmentation, since all the relevant tree edges are already covered by e. In addition, these paths include all tree edges because the edges added to the augmentation cover all tree edges. This proves (I).

Note that v adds e to the augmentation because the tree edge $\{v, p(v)\}$ is not covered yet. So v chooses e because it is the minimum weight edge e_v that covers $\{v, p(v)\}$. By Claim 1.6, it holds that $w(e_v) = \sum_{t' \in P} c(t')$ where $P = P_e$ is the path of tree edges between x and $p(v)$. This proves (II). (I) and (II) complete the proof that the cost of all the edges added to the augmentation is equal to the sum of costs of the edges in T. ◀

A similar proof shows that the cost of any augmentation of G' is at least the sum of costs of the edges of T. The idea is to map a subset of edges in the augmentation to disjoint paths in T, such that the union of the paths is the entire tree. We show that the weight of each edge is at least the sum of costs of the tree edges on the corresponding path, which proves the claim. The proof is in the full version. From the above and from Claim 1.7 we have that the cost of the augmentation obtained by the algorithm is smaller or equal to the cost of any augmentation of G', hence the solution obtained by the algorithm is optimal. This completes the proof of Lemma A.5. ◀

By Lemmas A.5 and A.4, Algorithm 1 finds an optimal augmentation A' in G' in $O(h)$ rounds. In the full version, we show that A' corresponds to a 2-approximation augmentation in G, and that building G' takes $O(h)$ rounds. This gives the following.

▶ **Theorem 1.1.** *There is a distributed 2-approximation algorithm for weighted TAP in the CONGEST model that runs in $O(h)$ rounds, where h is the height of the tree T.*

Schlegel Diagram and Optimizable Immediate Snapshot Protocol*

Susumu Nishimura

Dept. of Mathematics, Graduate School of Science, Kyoto University, Japan
susumu@math.kyoto-u.ac.jp

Abstract

In the topological study of distributed systems, the immediate snapshot is the fundamental computation block for the topological characterization of wait-free solvable tasks. However, in reality, the immediate snapshot is not available as a native built-in operation on shared memory distributed systems. Borowsky and Gafni have proposed a wait-free multi-round protocol that implements the immediate snapshot using more primitive operations, namely the atomic reads and writes.

In this paper, up to an appropriate reformulation on the original protocol by Borowsky and Gafni, we establish a tight link between each round of the protocol and a topological operation of subdivision using Schlegel diagram. Due to the fact shown by Kozlov that the standard chromatic subdivision is obtained by iterated subdivision using Schlegel diagram, the reformulated version is proven to compute the immediate snapshot in a topologically smoother way. We also show that the reformulated protocol is amenable to optimization: Since each round restricts the possible candidates of output to an iteratively smaller region of finer subdivision, each process executing the protocol can decide at an earlier round, beyond which the same final output is reached no matter how the remaining rounds are executed. This reduces the number of read and write operations involved in the overall execution of the protocol, relieving the bottleneck of access to shared memory.

1998 ACM Subject Classification C.2.4. Distributed Systems, F.1.1. Models of Computation

Keywords and phrases Immediate snapshot protocol, Schlegel diagram, chromatic subdivision, program specialization

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.22

1 Introduction

The snapshot models [1, 4, 6, 11] for the shared memory distributed system have been intensively studied for the analysis of solvability of distributed tasks in the wait-free (or more generalized failure) models. In particular, the immediate snapshot model [1, 4] and the iterated immediate snapshot mode [6] are central to the study. The (iterated) immediate snapshot protocol is modeled by a topological operation on simplicial complexes, namely, the (iterated) standard chromatic subdivision. This topological interpretation has boosted theoretical investigations on distributed systems using simplicial complexes. Most notably, Herlihy and Shavit have established the asynchronous computability theorem [13], which states that a distributed task is wait-free solvable in the asynchronous read-write shared memory model if and only if the task is expressed by a suitable pair of an iterated standard chromatic subdivision and a simplicial map.

* This work was supported by JSPS KAKENHI Grant Number 16K00016.

Although immediate snapshots are not natively supported in real distributed systems, the protocol proposed by Borowsky and Gafni [5] provides a wait-free implementation of it on asynchronous shared memory systems with atomic reads and writes. Their protocol is a multi-round protocol, where each individual process in a distributed system consisting of $n+1$ processes decides its snapshot only if it witnesses, for k-th round, $n+2-k$ different processes (including the process itself) that have written to shared memory.

In the present paper, we give yet another multi-round protocol for the immediate snapshot, reformulating the one by Borowsky and Gafni. Though both protocols work equally, the reformulated version has notable advantages over the original one.

1. The reformulated multi-round protocol induces a neat correspondence of each individual round with a topological construction, namely a subdivision using Schlegel diagram.
   Benavides and Rajsbaum [2] showed that each round of the protocol by Borowsky and Gafni does not simply subdivides the input complex but it produces an intermediate protocol complex that is not a (pseudo)manifold. They make use of 'collapsing' to describe how the protocol complex is transformed topologically at each protocol round, concluding that the standard chromatic subdivision is obtained as the final result. In contrast, the present paper shows that, up to reformulation, each protocol round exactly corresponds to a subdivision using Schlegel diagram. This series of corresponding subdivisions gives rise to the standard chromatic subdivision, due to a straightforward topological argument by Kozlov [17].
2. Due to the simpler topological structure, the reformulated version is amenable to mechanical optimization.
   In shared memory systems, shared memory access is a major bottleneck. Processes share a single shared memory module, which serializes simultaneous requests to handle them one at a time. The above mentioned multi-round protocols for the immediate snapshot involve read and write requests multiplied by the number of rounds to be performed (and also by the number of processes). The multiplied requests to the shared memory thus can cause performance degradation due to memory contention.
   For each concrete implementation of a protocol that makes use of the immediate snapshot, the reformulated version of the immediate snapshot protocol can be optimized to issue a lesser number of memory requests so that each process decides its output value at an earlier round, beyond which any execution path converges to a single unique output. This refinement is possible because the reformulated protocol iteratively subdivides the protocol complex at each round, narrowing down the candidates of output to those in a smaller region of finer subdivision. Thus the output can be decided as soon as the possible outputs have been narrowed down to a singleton set. This optimizes the protocol to perform a lesser number of shared memory access, where the optimized code can be mechanically derived for each concrete instance.
   With this optimization technique, the author believes that the immediate snapshot model, which has been studied mostly of theoretical concern, can also serve as a fundamental construct for wait-free distributed programming in a more practical context.

Related Work

In [17], Kozlov has proven that the standard chromatic subdivision is indeed a subdivision by showing the standard chromatic subdivision is obtained by the iterated subdivision using Schlegel diagram. He also argued that the transient complexes that appear in the intermediate steps of iterated subdivision can be given computational interpretation, but

his interpretation combines atomic writes with immediate scan operations. In contrast, the present paper gives the computational interpretation solely with atomic reads and writes, establishing the exact correspondence of each round of the reformulated immediate snapshot protocol with subdivision using Schlegel diagram.

Benavides and Rajsbaum [2] studied the topological structure induced by the multi-round immediate snapshot protocol of Borowsky and Gafni. They observed that the series of shared memory reads and writes involved in each single round of the protocol generates a protocol complex that augments the standard chromatic subdivision with extra simplexes. Due to those extras, the protocol complexes are neither a subdivision of the input complex nor a (pseudo)manifold in general. They analyzed the topological structure of the protocol complexes in detail and presented a topological model, in which subsequent rounds of the protocol collapse those extra simplexes to end up with the standard chromatic subdivision. This collapsing series of complexes, though topologically insightful, does not explain well the correspondence to the underlying operational (execution) model. Assuming the colored simplicial topological model, where every simplex is a collection of vertexes of distinct colors (process ids), there can be no operational counterpart to collapsing of a simplex: Since a simplex is always collapsed to a degenerate simplex of strictly smaller dimension (i.e., of fewer colors), it absurdly indicates that such an operation would shrink a process group into a strictly smaller one (even if no process in the group crashed). In the present paper, we reformulate the multi-round protocol by Borowsky and Gafni so that each round is precisely a subdivision using Schlegel diagram, providing a simpler topological model. The neat correspondence between the topological model and the operational model also allows us to optimize the protocol for reduced shared memory access. (See Section 4.)

Hoest and Shavit [14] proposed the nonuniform iterated immediate snapshot model, a refinement of the immediate snapshot model, for the purpose of a precise analysis of protocol complexity. The nonuniform iterated immediate snapshot topologically corresponds to the iterated nonuniform chromatic subdivision, which generalizes the iterated standard chromatic subdivision so that individual simplexes are allowed to be subdivided different numbers of times. Their nonuniform model may also be applied to protocol optimization. That is, a task solvable by a protocol in the uniform model would be substituted by a protocol in the nonuniform model that solves the same task with a coarser iterated nonuniform subdivision. However, it seems nontrivial in general to find an optimal nonuniform protocol. The reformulated multi-round protocol in the present paper, due to the smooth correspondence between the topological and operational models, allows mechanical optimization on any protocol given in the uniform model.

Outline

The rest of the paper is organized as follows. Section 2 describes the shared memory distributed computing model and the wait-free multi-round protocol by Borowsky and Gafni and reviews the topological theory concerning wait-free solvability of distributed tasks. Section 3 proposes a reformulation of the multi-round protocol by Borowsky and Gafni. We prove the reformulated protocol also computes the immediate snapshot, showing that each round of the reformulated protocol precisely corresponds to a subdivision using Schlegel diagram. In Section 4, we further argue that the reformulated protocol is amenable to optimization. We show that, for each concrete protocol that solves a task using the iterated immediate snapshot, the protocol can be optimized to decide the final output at an earlier round, as soon as the collection of possible outputs to be reached is narrowed down to a singleton set. Finally, Section 5 concludes the paper.

2 Distributed Computing and Topological Model

Throughout the paper we consider a distributed system of $n+1$ faulty processes, which have process ids numbered 0 through n. We assume the asynchronous read-write shared memory model, where the distributed system has a shared memory consisting of single-writer, multi-reader atomic registers and processes can communicate solely by atomic reads and writes on these registers. We further assume that each process i receives its initial private input value through the variable v_i, which is local to the process.

The rest of this section is devoted to give an overview of the immediate snapshot model and the basics of combinatorial topology related to the topological theory of wait-free solvability of distributed tasks. For a more complete exposition on the subject, see [11, 16].

2.1 Immediate snapshot in the read-write model

Borowsky and Gafni proposed the wait-free implementation of the immediate snapshot in the read-write shared memory model [5]. Algorithm 1 gives the Borowsky-Gafni protocol in a recursive style implementation by Gafni and Rajsbaum [10]. The protocol is a multi-round protocol, where the series of recursive calls $\mathsf{IS}(n)$, $\mathsf{IS}(n-1)$, ..., $\mathsf{IS}(0)$ correspond to $n+1$ multiple rounds and the processes communicate through a series of $n+1$ shared memory arrays mem_d ($n \geq d \geq 0$), each of which consists of $n+1$ registers indexed 0 through n. Each process i, per each recursive call $\mathsf{IS}(d)$ for the $(n+1-d)$-th round, invokes the *write&scan* operation WScan on the shared memory array mem_d. When WScan is invoked by process i, it first writes the private value v_i of the process i to the register $mem_d[i]$, then scans the view, i.e., the set of values that have been written in the array mem_d, and returns the view paired with the process id. The view is gathered by collect, which reads the registers of the array in an unspecified order. If process i witnesses $d+1$ values in the view, it returns the pair of process id and the view as the result of immediate snapshot, terminating recursion; Otherwise, it continues recursive call $\mathsf{IS}(d-1)$ for another round.

We notice that the view returned by WScan per each round is simply discarded, when not sufficiently many values are witnessed.

2.2 Simplicial complexes and subdivisions

Let V be a set of vertexes. A *simplex* σ is a finite subset of V. The dimension of σ, denoted by $\dim(\sigma)$, is given by $|\sigma|-1$. A simplex σ of dimension d is called a *d-simplex*. In particular, the empty simplex $\emptyset$ is a (-1)-simplex. A simplex σ is called a *face* of τ if $\sigma \subseteq \tau$ and is particularly called a *proper* face if the inclusion is strict.

A *simplicial complex* (or a *complex* for short) $\mathcal{C}$ is a set of simplexes such that $\sigma \in \mathcal{C}$ and $\tau \subseteq \sigma$ implies $\tau \in \mathcal{C}$. The dimension of $\mathcal{C}$, written $\dim(\mathcal{C})$, is the maximum dimension of simplexes contained in $\mathcal{C}$. A complex of dimension d is also called a *d-complex*. We write $V(\mathcal{C})$ for the set of vertexes contained in $\mathcal{C}$. A complex $\mathcal{D}$ is called a *subcomplex* of $\mathcal{C}$, if $\mathcal{D} \subseteq \mathcal{C}$. The *$k$-skeleton* of a complex $\mathcal{C}$, written $\mathrm{skel}^{(k)}\,\mathcal{C}$, is the maximum subcomplex of $\mathcal{C}$ of dimension k or less, namely, $\mathrm{skel}^{(k)}\,\mathcal{C} = \{\sigma \in \mathcal{C} \mid \dim(\sigma) \leq k\}$.

A *facet* σ of $\mathcal{C}$ is a maximal simplex, i.e., σ is not a proper face of any $\tau \in \mathcal{C}$. We write $\overline{\sigma}$ for a complex whose sole facet is the simplex σ, i.e., $\overline{\sigma} = \{\tau \mid \tau \subseteq \sigma\}$. A complex is called *pure*, if all its facets have the same dimension.

Whenever $\sigma \cap \tau = \emptyset$, we write $\sigma * \tau$ for the *join* of the simplexes σ and τ, namely the union $\sigma \cup \tau$. Likewise, whenever $V(\mathcal{C}) \cap V(\mathcal{D}) = \emptyset$, we define the *join* of $\mathcal{C}$ and $\mathcal{D}$ by $\mathcal{C} * \mathcal{D} = \{\sigma * \tau \mid \sigma \in \mathcal{C}, \tau \in \mathcal{D}\}$. The *star* of a vertex $v \in \mathcal{C}$ is defined by $\mathrm{St}(v, \mathcal{C}) = \{\tau \in \mathcal{C} \mid \{v\} \cup \tau \in \mathcal{C}\}$, which is the maximum subcomplex of $\mathcal{C}$ whose every facet contains v.

Algorithm 1 Multi-round write&scan code for the process i.

procedure WScan(d)
  $mem_d[i] \leftarrow v_i$
  $view \leftarrow \mathsf{collect}(mem_d)$
  return $(i, view)$

procedure IS(d)
  $(i, view) \leftarrow \mathsf{WScan}(d)$
  if $|view| = d + 1$ **then return** $(i, view)$
  else IS$(d-1)$

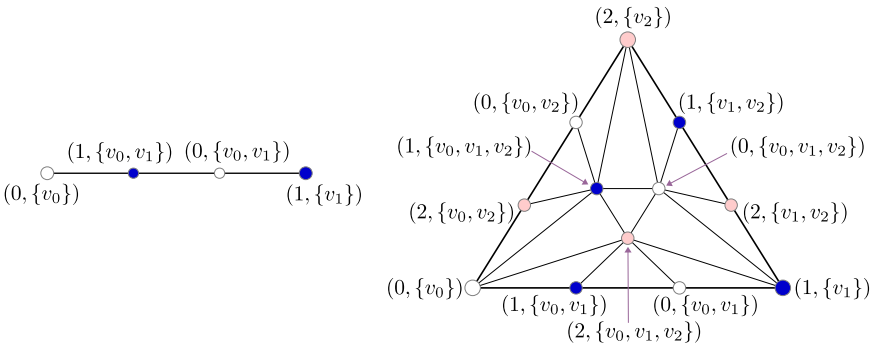


Figure 1 The standard chromatic subdivisions on 1-simplex and 2-simplex.

Throughout the paper, complexes are assumed to be pure. Also, we solely consider the so-called *chromatic* simplexes and complexes. Suppose we have a *coloring function* $\mathrm{color} : V \to \{0, \ldots, n\}$. A simplex σ is chromatic if $\mathrm{color}(v) = \mathrm{color}(v')$ implies $v = v'$ for every $v, v' \in \sigma$; A complex $\mathcal{C}$ is chromatic if every simplex $\sigma \in \mathcal{C}$ is chromatic.

A *simplicial map* is a total vertex map $\mu : V(\mathcal{C}) \to V(\mathcal{D})$ such that $\mu(\sigma) \in \mathcal{D}$ for every $\sigma \in \mathcal{C}$. Every simplicial map must be *color-preserving*, i.e., $\mathrm{color}(v) = \mathrm{color}(\mu(v))$ for every $v \in V(\mathcal{C})$.

A *subdivision* of a complex $\mathcal{C}$, written $\mathrm{Div}\,\mathcal{C}$, is a finer complex obtained by dividing each simplex of the complex into smaller pieces of (chromatic) simplexes. (In this paper, though we occasionally refer to geometric presentation of subdivisions, we are solely concerned with combinatorial definition of subdivision in formality. For example, $\overline{\sigma}$ should be understood as the trivial subdivision, which does not geometrically refine σ at all.) For a simplex $\tau \in \mathrm{Div}\,\mathcal{C}$, we write $\mathrm{Carr}(\tau, \mathcal{C})$ for the *carrier* of τ, namely, the smallest simplex $\sigma \in \mathcal{C}$ such that $\tau \in \mathrm{Div}\,\sigma$.

A subdivision induces a *parent map* π, which carries each vertex of the subdivision to the unique vertex $\pi(v)$ of matching color in its carrier. That is, for $v \in V(\mathrm{Div}\,\mathcal{C})$, $\pi(v)$ is the unique vertex of $\mathcal{C}$ such that $\pi(v) \in \mathrm{Carr}(\{v\}, \mathcal{C})$ and $\mathrm{color}(\pi(v)) = \mathrm{color}(v)$. The parent map $\pi : V(\mathrm{Div}\,\mathcal{C}) \to V(\mathcal{C})$ is a color-preserving simplicial map.

2.3 Distributed task and asynchronous computability

A (colored) *task* for a distributed system with $n+1$ faulty processes is defined by a triple $(\mathcal{I}, \mathcal{O}, \Phi)$, where $\mathcal{I}$ is an input complex (of dimension n), $\mathcal{O}$ is an output complex (of dimension n), and $\Phi : \mathcal{I} \to 2^{\mathcal{O}}$ is a *carrier map*, a monotonic function that maps every input simplex $\sigma \in \mathcal{I}$ to an output subcomplex $\Phi(\sigma) \subseteq \mathcal{O}$ such that $\mathrm{color}(\sigma) = \bigcup\{\mathrm{color}(\tau) \mid \tau \in \Phi(\sigma)\}$.

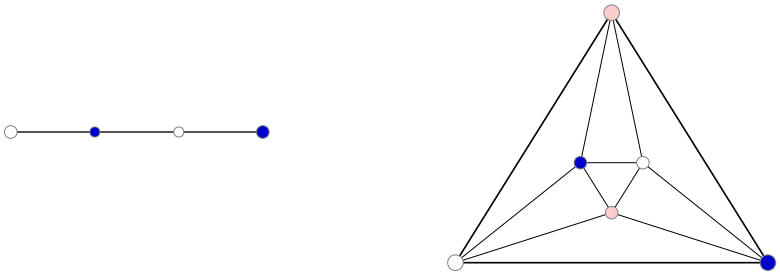


Figure 2 Schlegel diagrams for 1-simplex and 2-simplex.

The asynchronous computability theorem [13] gives a topological characterization for the class of tasks that are wait-free solvable in the shared memory read-write model. The primary topological tool employed in the theorem is subdivision on simplicial complexes, especially the *standard chromatic subdivision*. See Figure 1 for geometric presentation of the standard chromatic subdivision on 1-simplex $\{v_0, v_1\}$ and 2-simplex $\{v_0, v_1, v_2\}$: The standard chromatic subdivision refines each d-simplex σ ($d > 0$) by introducing new $d+1$ vertexes (of different colors) at antiprismatic positions displaced from the barycenter of the simplex. Combinatorially, as indicated in Figure 1, each vertex of color i contained in the subdivision is designated by a pair (i, σ), where σ is a nonempty subset of the original simplex and $i \in \mathrm{color}(\sigma)$. (Herein and after we will depict the vertexes for different processes with colors white, dark blue, and light red to indicate processes of id 0, 1, and 2, respectively.) Furthermore, a set $\{(i_0, \tau_0), \ldots, (i_d, \tau_d)\}$ of vertexes of distinct colors forms a d-simplex of the subdivision if and only if the subsets of the original simplex are linearly ordered (as $\tau_0 \subseteq \cdots \subseteq \tau_d$, up to appropriate permutation).

In what follows, let us write $\mathrm{Ch}\,\mathcal{C}$ for the standard chromatic subdivision applied to every simplex in $\mathcal{C}$ and also write $\mathrm{Ch}^m\mathcal{C}$ ($m \geq 0$) for the m-iterated application of Ch on $\mathcal{C}$.

The multi-round protocol for the immediate snapshot (Algorithm 1) in effect implements the standard chromatic subdivision [2]: Every non-faulty process i executing the protocol returns a vertex (i, τ) of the subdivision.

▶ **Theorem 1** (Theorem 5.29 and Corollary 5.31 of [13]). *A colored task $(\mathcal{I}, \mathcal{O}, \Phi)$ has a wait-free protocol in the asynchronous shared memory read-write model if and only if there exists a subdivision* $\mathrm{Div}\,\mathcal{I}$ *and a* decision map δ*, which is a color-preserving simplicial map $\delta : V(\mathrm{Div}\,\mathcal{I}) \to V(\mathcal{O})$ such that $\delta(\sigma) \in \Phi(\mathrm{Carr}(\sigma, \mathcal{I}))$ for every $\sigma \in \mathrm{Div}\,\mathcal{I}$.*

In particular, the subdivision Div *can be taken* Ch^K *for some $K \geq 0$.*

3 Immediate Snapshot as Iterated Subdivision Using Schlegel Diagram

This section presents a multi-round protocol for the immediate snapshot, by reformulating the protocol by Borowsky and Gafni [5]. We show that each round of the protocol corresponds exactly to a subdivision using Schlegel diagram.

3.1 Schlegel diagram and subdivision

A *Schlegel diagram* is a projection of a polytope onto one of its facets [19]. In the present paper, we are solely concerned with Schlegel diagrams on cross-polytopes. The Schlegel diagram on $(d+1)$-dimensional cross-polytope, which consists of $2(d+1)$ vertexes, gives

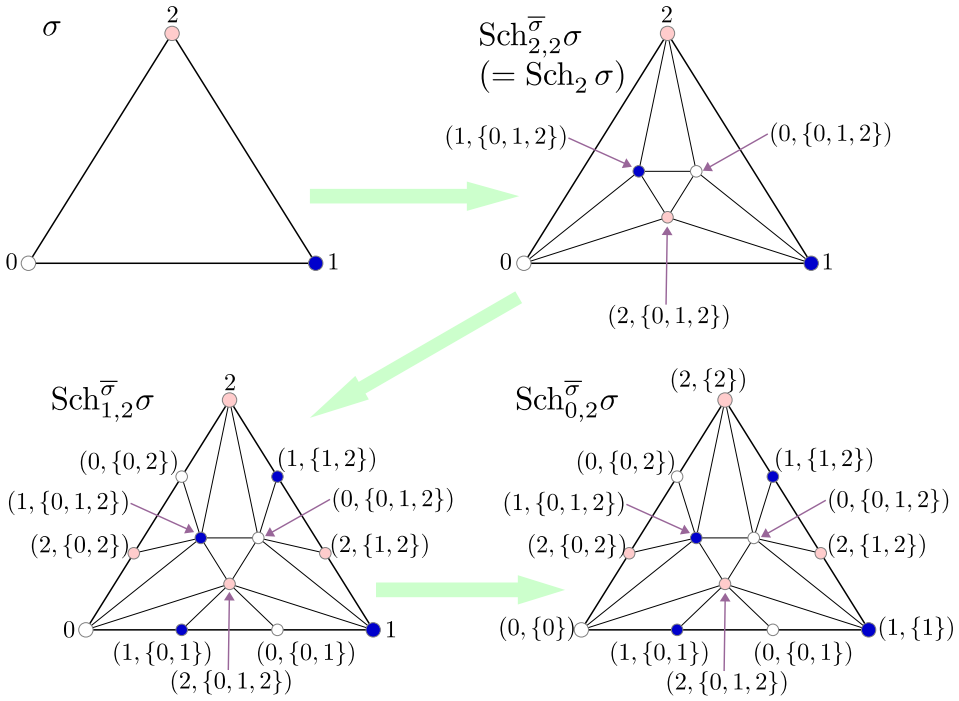


Figure 3 Standard chromatic subdivision on $\sigma = \{0,1,2\}$ using Schlegel diagrams.

a subdivision of d-simplex. Figure 2 shows subdivisions of 1-simplex and 2-simplex by Schlegel diagrams, where the former is derived from the quadrilateral and the latter from the octahedron. (Note that the standard chromatic subdivision is a refinement of Schlegel diagram in general, but they coincide for 1-simplexes.)

The complex of Schlegel diagram that subdivides a d-simplex $\sigma = \{v_0, \ldots, v_d\}$, denoted by $\mathrm{Sch}_d\,\sigma$, is formally defined as follows:

$$\mathrm{Sch}_d\,\sigma = \bigcup \{ \overline{\{v_i \mid i \in \sigma \setminus I\} * \{(i,\sigma) \mid i \in I\}} \mid \emptyset \subsetneq I \subseteq \{0,\ldots,d\} \},$$

where each (i,σ) is a new vertex introduced for subdivision, with coloring $\mathrm{color}((i,\sigma)) = i$. Geometrically, the Schlegel diagram subdivides a d-simplex σ into smaller facets, namely, the central facet $\{(0,\sigma),\ldots,(d,\sigma)\}$, which is solely comprised of the new vertexes, and other facets, each of which shares a lower dimensional simplex with the central facet. (See Figure 3 for the subdivision of 2-simplex.) Notice that σ is no more a face of Schlegel subdivision $\mathrm{Sch}_d\,\sigma$, as it does not share any simplex with the central facet. When $\dim(\sigma) \neq d$, we define $\mathrm{Sch}_d\,\sigma$ by a trivial subdivision, i.e., $\mathrm{Sch}_d\,\sigma = \overline{\sigma}$.

Kozlov [17] has shown that the standard chromatic subdivision can be obtained by a series of Schlegel diagram that subdivides simplexes in the order of decreasing dimension. To put it formal, let $\mathcal{D}$ be any subdivision of a d-complex $\mathcal{C}$ such that $\mathcal{C} \cap \mathcal{D} = \mathrm{skel}^{(k)}\,\mathcal{C}$, meaning that $\mathcal{D}$ subdivides simplexes of $\mathcal{C}$ up to dimension $k+1$ and higher but no simplexes of lower dimension. Let us write $\mathrm{Sch}^{\mathcal{C}}_k\mathcal{D}$ for the subdivision of $\mathcal{D}$ applied to every k-simplex of $\mathcal{C} \cap \mathcal{D}$ by Schlegel diagram, namely,

$$\mathrm{Sch}^{\mathcal{C}}_k\mathcal{D} = \{\tau * \sigma' \mid \tau * \sigma \in \mathcal{D} \text{ and } \sigma' \in \mathrm{Sch}_k\,\sigma \text{ for some } \sigma \in \mathcal{C} \cap \mathcal{D}\}.$$

Let us also write $\mathrm{Sch}^{\mathcal{C}}_{h,j}$ for the composition $\mathrm{Sch}^{\mathcal{C}}_h \circ \mathrm{Sch}^{\mathcal{C}}_{h+1} \circ \cdots \circ \mathrm{Sch}^{\mathcal{C}}_j$ $(0 \leq h, j \leq d)$ of subdivisions on simplexes in the order of decreasing dimension. (When $h > j$, $\mathrm{Sch}^{\mathcal{C}}_{h,j}$ denotes the trivial subdivision.)

Algorithm 2 Multi-round write&oblivious scan code for the process i.

```
procedure WOScan(d)
    mem_d[i] ← v_i
    view ← collect(mem_d)
    if |view| = d + 1 then return (i, view)
    else return v_i
procedure IS'(d)
    u ← WOScan(d)
    if u ≠ v_i then return u
    else IS'(d − 1)
```

▶ **Theorem 2** ([17]). *For any pure complex $\mathcal{C}$ of dimension d, $\operatorname{Ch}\mathcal{C} = \operatorname{Sch}^{\mathcal{C}}_{0,d}\mathcal{C}$.*

Figure 3 shows how the standard chromatic subdivision on a 2-simplex $\sigma = \{0,1,2\}$ with $\operatorname{color}(i) = i$ for every $i \in \{0,1,2\}$ is obtained by the series of subdivisions using Schlegel diagram. For example, the 2-simplex $\{0,1\} * \{(2,\{0,1,2\})\}$ in $\operatorname{Sch}^{\overline{\sigma}}_{2,2}$ is further refined in the next step of subdivision, say, the 1-simplex $\{0,1\}$ is subdivided into three parts $\{0,(1,\{0,1\})\}$, $\{(0,\{0,1\}),(1,\{0,1\})\}$, $\{(0,\{0,1\}),1\}$, which are each *joined* with $\{(2,\{0,1,2\})\}$.

3.2 The immediate snapshot protocol with oblivious scan

Algorithm 2 gives a multi-round immediate snapshot protocol, which reformulates Algorithm 1 in Section 2.1. It is easy to see that they are indeed equivalent protocols in different presentations. Remember that in Algorithm 1 the view information collected at each particular round is discarded unless the view witnesses the expected number of writes. Algorithm 2 just makes this explicit by employing the *write&oblivious scan* operation WOScan on shared memory array, in place of write&scan operation. When process i calls WOScan(d) and the view does not witness d writes, WOScan(d) returns v_i, discarding the view collected at the scan phase.

With this reformulation, we can prove that the protocol computes the standard chromatic subdivision, with an exact correspondence of a write&oblivious scan operation at a particular round with Schlegel diagram.

▶ **Lemma 3.** *Suppose WOScan(d) has ever been called by $d+1$ distinct processes. If $\sigma = \{v_0,\dots,v_d\}$ is the collection of private values that have been assigned to the $d+1$ processes and τ is the set of results returned by non-faulty processes, then $\tau \in \operatorname{Sch}_d \sigma$.*

Proof. When $d+1$ processes invoked WOScan(d) and none of them were faulty, at least one process witnesses the writes by all the $d+1$ processes in its view. This means $\tau \setminus \sigma \neq \emptyset$ and thus $\tau \in \operatorname{Sch}_d \sigma$. When some of the processes were faulty, $\dim(\tau) < d$ and $\tau \in \operatorname{Sch}_d \sigma$. ◀

▶ **Lemma 4.** *Consider an execution of the protocol IS'(n) by $n+1$ processes, in which each process i has started with its own initial private value v_i and has either successfully returned a result or crashed. Let σ_d $(n \geq d \geq 0)$ denote the set of results that have been successfully returned by a call WOScan(d) by some process in the execution. Also, let τ_d $(n \geq d \geq 0)$ denote the set of values that have been returned by a call IS'(k) for some k greater than $d-1$. We define $\sigma_{n+1} = \{v_0,\dots,v_n\}$ and $\tau_{n+1} = \emptyset$.*

Then the following properties hold for every d $(n+1 \geq d \geq 0)$.

- **(i)** $\dim(\sigma_d \cap \sigma_{n+1}) < d$;
- **(ii)** $\tau_d = \tau_{d+1} * (\sigma_d \setminus \sigma_{n+1})$ *and* $\tau_d \cap \sigma_{n+1} = \emptyset$;
- **(iii)** $\tau_{d+1} * \sigma_d \in \operatorname{Sch}^{\overline{\sigma_{n+1}}}_{d,n} \sigma_{n+1}$.

Algorithm 3 The generic code for the process i, using iterated immediate snapshot.

```
procedure WOScan(k, d)
    mem_{k,d}[i] ← v_i
    view ← collect(mem_{k,d})
    if |view| = d + 1 then return (i, view)
    else return v_i
procedure IIS(k, d)
    u ← WOScan(k, d)
    if u ≠ v_i then
        if k = K then return δ(u)
        else v_i ← u; IIS(k + 1, n)
    else IIS(k, d − 1)
```

Proof. The property (i) follows from lemma 3 by induction on d. The property (ii) immediately follows from the definition by an inductive argument.

Let us show (iii) by induction on d. For the base case $d = n$, since $\sigma_n \in \mathrm{Sch}_n \sigma_{n+1}$ by lemma 3, we have $\tau_{n+1} * \sigma_n = \sigma_n \in \mathrm{Sch}_{n,n}^{\overline{\sigma_{n+1}}} \sigma_{n+1}$. For the inductive step, assume $\tau_{d+2} * \sigma_{d+1} \in \mathrm{Sch}_{d+1,n}^{\overline{\sigma_{n+1}}} \sigma_{n+1}$. By lemma 3 we have $\sigma_d \in \mathrm{Sch}_d(\sigma_{d+1} \cap \sigma_{n+1})$. Since $\sigma_{d+1} \cap \sigma_{n+1} \in \overline{\sigma_{n+1}}$, we have $\tau_{d+1} * (\sigma_{d+1} \cap \sigma_{n+1}) = \tau_{d+2} * (\sigma_{d+1} \setminus \sigma_{n+1}) * (\sigma_{d+1} \cap \sigma_{n+1}) = \tau_{d+2} * \sigma_{d+1} \in \mathrm{Sch}_{d+1,n}^{\overline{\sigma_{n+1}}} \sigma_{n+1}$ by property (ii) and the induction hypothesis. Hence $\tau_{d+1} * \sigma_d \in \mathrm{Sch}_d^{\overline{\sigma_{n+1}}}(\mathrm{Sch}_{d+1,n}^{\overline{\sigma_{n+1}}} \sigma_{n+1}) = \mathrm{Sch}_{d,n}^{\overline{\sigma_{n+1}}} \sigma_{n+1}$. ◀

▶ **Theorem 5.** *Suppose $n + 1$ processes executed the protocol IIS′(n) with the set $\sigma = \{v_0, \ldots, v_n\}$ of initial private inputs. If τ is the set of results returned by non-faulty processes, $\tau \in \mathrm{Ch}\,\sigma$.*

Proof. Let σ_d's and τ_d's denote the sets as defined in lemma 4. Then, $\sigma = \sigma_{n+1}$ and $\tau = \tau_1 * \sigma_0$. By lemma 4(iii) and theorem 2, we have $\tau = \tau_1 * \sigma_0 \in \mathrm{Sch}_{0,n}^{\overline{\sigma}} \sigma = \mathrm{Ch}\,\sigma$. ◀

4 The Generic Protocol for Solving Tasks and Its Optimization

This section gives a generic protocol for solving a task on read-write shared memory distributed system, using the immediate snapshot protocol presented in the previous section, and discusses how, for each concrete instance, the generic protocol can be optimized to reduce shared memory access.

4.1 A generic protocol via iterated immediate snapshot

By the asynchronous computability theorem, for any wait-free solvable task, we have a protocol $(\mathcal{I}, \mathcal{O}, \delta \circ \mathrm{Ch}^K)$ that implements the task, where the carrier map is given by a pair of the full-information protocol of the K-iterated standard chromatic subdivision Ch^K (by means of the iterated use of the multi-round immediate snapshot protocol) and a decision map $\delta : V(\mathrm{Ch}^K) \to V(\mathcal{O})$. Without loss of generality, we may assume $K \geq 1$.

Algorithm 3 gives the code that implements the protocol $(\mathcal{I}, \mathcal{O}, \delta \circ \mathrm{Ch}^K)$ in the generic form. Each process i initiates the protocol execution by invoking IIS$(1, n)$, where its initial private input is passed through the variable v_i. Throughout the entire protocol execution, each process goes through a series of shared memory arrays $mem_{k,d}$ $(1 \leq k \leq K,\ n \geq d \geq 0)$. For each

recursive call $\mathsf{IIS}(k,d)$, process i computes the $(n-d+1)$-th round of the k-th iteration of the multi-round immediate snapshot protocol, by invoking the write&oblivious scan $\mathsf{WOScan}(k,d)$ on the array $mem_{k,d}$. Each round corresponds to a single step of subdivision on d-simplexes using Schlegel diagram, for the k-th iteration of standard chromatic subdivision. When the multi-round execution by process i finishes the last iteration of chromatic subdivision (i.e., $k=K$), the protocol returns $\delta(u)$ as the output, where u is a vertex of the K-iterated standard chromatic subdivision.

The following is a corollary to Theorem 5.

▶ **Theorem 6.** *Let $(\mathcal{I},\mathcal{O},\delta\circ\mathrm{Ch}^K)$ be a protocol for $n+1$ processes that solves a task. Suppose each process i starts with a private input value v_i such that $\sigma=\{v_0,\ldots,v_n\}\in\mathcal{I}$ and executes the protocol by invoking $\mathsf{IIS}(1,n)$. If τ is the set of results returned by non-faulty processes, $\tau\in\delta(\mathrm{Ch}^K\sigma)$.*

4.2 Protocol optimization for reduced memory access

Algorithm 3 gives a generic protocol, but a concrete instance of it often contains redundant memory access. Below we argue that each instance of the generic protocol can be mechanically optimized to skip the redundant access, by applying the technique of program specialization. (Program specialization is a source-level program optimization technique, also known as *partial evaluation* [15]. See Appendix A for a brief overview.)

To see how the protocol is optimized, let us consider a particular instance of the generic protocol that solves a renaming task [5] for 3 processes,[1] where the 3 processes, starting with initial assignment $v_0=0$, $v_1=1$, $v_2=2$, respectively, decide on different names taken from $\{0,1,2,3,4\}$. The protocol is given by $(\mathcal{I},\mathcal{O},\delta\circ\mathrm{Ch}^2)$, where $\mathcal{I}=\overline{\{0,1,2\}}$ is the input complex, $\mathcal{O}=\{\tau\mid\tau\subseteq\{0,1,2,3,4\},\dim(\tau)\le 2\}$ is the output complex of differently renamed vertexes, and $\delta:V(\mathrm{Ch}^2\mathcal{I})\to V(\mathcal{O})$ is the decision map defined with the corresponding parent map $\pi_{2,1}:V(\mathrm{Ch}^2\mathcal{I})\to V(\mathrm{Ch}\,\mathcal{I})$ as given below:

$$\delta(w)=\begin{cases}
4 & \text{if } \pi_{2,1}(w)=(0,\{0,1,2\}),\\
3 & \text{if } \mathrm{Carr}(w,\mathrm{Ch}\,\mathcal{I})\not\supseteq\{(1,\{0,1,2\}),(2,\{0,1,2\})\}\\
 & \text{and } \pi_{2,1}(w)=(i,\{0,1,2\}) \text{ for some } i\in\{1,2\},\\
2 & \text{if either } \mathrm{color}(w)=1 \text{ and } \mathrm{Carr}(w,\mathrm{Ch}\,\mathcal{I})\supseteq\{(1,\{0,1,2\}),(2,\{0,1,2\})\}\\
 & \text{or } \pi_{2,1}(w)=(i,\{i,j\}) \text{ for some } i,j\in\{0,1,2\} \text{ such that } i<j,\\
1 & \text{if either } \mathrm{color}(w)=2 \text{ and } \mathrm{Carr}(w,\mathrm{Ch}\,\mathcal{I})\supseteq\{(1,\{0,1,2\}),(2,\{0,1,2\})\}\\
 & \text{or } \pi_{2,1}(w)=(i,\{i,j\}) \text{ for some } i,j\in\{0,1,2\} \text{ such that } i>j,\\
0 & \text{if } \pi_{2,1}(w)=(i,\{i\}) \text{ for some } i\in\{0,1,2\}.
\end{cases}$$

In the definition above, it is assumed that each vertex is appropriately colored according to the context. In particular, as every simplex $\tau\in\mathcal{O}$ of renamed processes is colored, $V(\mathcal{O})$ comprises 15 vertexes, namely, 3 differently colored vertexes per each name taken from $\{0,1,2,3,4\}$. Accordingly, as δ is a color-preserving simplicial map, the renamed output $\delta(w)$ for each $w\in V(\mathcal{O})$ is tacitly given the matching color, i.e., $\mathrm{color}(w)$.

Figure 4 shows how the decision map δ assigns an output to each vertex of $\mathrm{Ch}^2\mathcal{I}$. For those vertexes whose outputs are left unspecified in the figure, it should be understood that such a vertex w receives the output $\delta(\pi_{2,1}(w))$, namely the same output as the parent vertex $\pi_{2,1}(w)\in\mathrm{Ch}\,\mathcal{I}$ does. For instance, the white vertex (of process 0) of the very central simplex

[1] This particular instance of renaming task is taken from [11, Chapter 12].

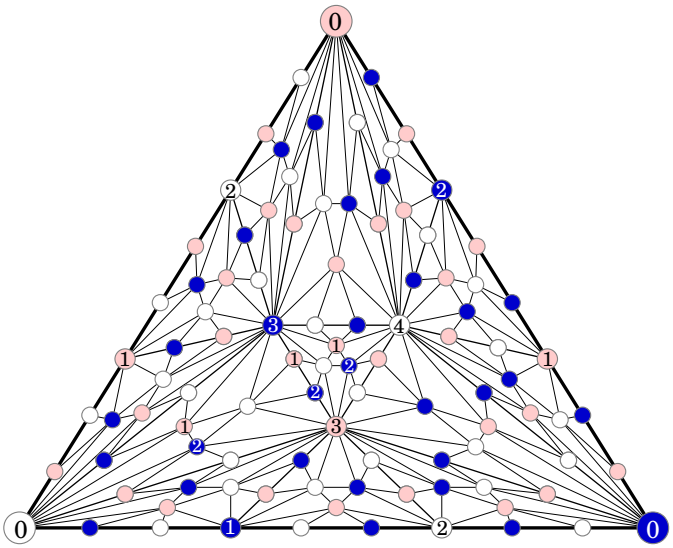


Figure 4 Decision map δ for renaming.

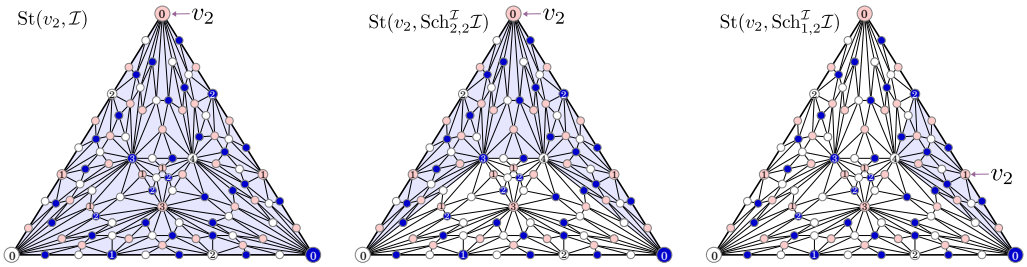


Figure 5 Descendant vertexes in the iterated subdivisions (up to the second iteration, where process 2 is ready for decision).

of the subdivision is assigned the output 4 by δ, as its parent is $(0, \{0, 1, 2\})$, the white vertex introduced by the first subdivision using Schlegel diagram.

Let us consider an execution of the generic protocol (Algorithm 3) for the renaming task. In the execution, each process i executes a chain of recursive calls of IIS, where each recursive call updates v_i to a vertex of a finer subdivision. For instance, consider the following particular recursive call chain for process 2:

$$\begin{aligned} 2 \in \mathcal{I} &\xrightarrow{\mathsf{IIS}(1,2)} 2 \in \mathrm{Sch}^{\mathcal{I}}_{2,2}\mathcal{I} \xrightarrow{\mathsf{IIS}(1,1)} (2, \{1, 2\}) \in \mathrm{Sch}^{\mathcal{I}}_{1,2}\mathcal{I} \\ &\xrightarrow{\mathsf{IIS}(2,2)} (2, \{1, 2\}) \in \mathrm{Sch}^{\mathrm{Ch}\,\mathcal{I}}_{2,2}(\mathrm{Ch}\,\mathcal{I}) \\ &\xrightarrow{\mathsf{IIS}(2,1)} (2, \{(0, \{0, 1, 2\}), (2, \{1, 2\})\}) \in \mathrm{Sch}^{\mathrm{Ch}\,\mathcal{I}}_{1,2}(\mathrm{Ch}\,\mathcal{I}), \end{aligned}$$

where each transition $u \in \mathrm{Div}\,\mathcal{I} \xrightarrow{\mathsf{IIS}(k,d)} u' \in \mathrm{Div}'\mathcal{I}$ indicates that a recursive call $\mathsf{IIS}(k, d)$ updates v_2 from u to u', which are the vertexes of subdivisions $\mathrm{Div}\,\mathcal{I}$ and $\mathrm{Div}'\mathcal{I}$, respectively. Process 2 terminates the execution of the protocol with a final output $\delta((2, \{(0, \{0, 1, 2\}), (2, \{1, 2\})\})) = 1$.

This recursive call chain, however, could have decided the final output at an earlier stage of recursion, because the vertex $v_2 \in \mathrm{Div}\,\mathcal{I}$ at each recursive call can only be updated to a

Algorithm 4 An optimized generic code for process i.

```
procedure IIS(k, d)
    if δ(π^{-1}(v_i)) = {u} for some u ∈ V(O) then return u
    u ← WOScan(k, d)
    if u ≠ v_i then v_i ← u; IIS(k + 1, n)
    else IIS(k, d − 1)
```

vertex (of matching color) covered by $\mathrm{St}(v_2, \mathrm{Div}\,\mathcal{I})$ by the successive recursive calls. Figure 5 illustrates the first few steps, where the shaded part indicates the simplexes of $\mathrm{Ch}^2\mathcal{I}$ covered by the star at each recursive step. Initially, when the input complex $\mathcal{I}$ is not yet subdivided, $\mathrm{St}(v_2, \mathcal{I})$ covers all the vertexes of $\mathrm{Ch}^2\mathcal{I}$; After the first recursive call $\mathsf{IIS}(1, 2)$, $\mathrm{St}(v_2, \mathrm{Sch}^{\mathcal{I}}_{2,2}\mathcal{I})$ covers fewer vertexes in a smaller region but they do not agree with the outputs carried by δ (they can be either 0 or 1); After the second recursive call $\mathsf{IIS}(1, 1)$, $\mathrm{St}(v_2, \mathrm{Sch}^{\mathcal{I}}_{1,2}\mathcal{I})$ covers even fewer vertexes and they are all carried to the same output vertex 1 by δ. Hence, as soon as v_2 is updated to the vertex $(2, \{1, 2\})$ by the recursive call $\mathsf{IIS}(1, 1)$, we can decide the output of process 2 by $\delta((2, \{1, 2\})) = 1$, skipping the remaining recursive calls.

By the observation so far, we can see that the generic protocol (Algorithm 3) can be further optimized to perform fewer shared memory operations by skipping redundant recursive calls: Once a process has reached to a point where a sole final output is determined by the decision map δ, the remaining recursive calls can be skipped. To make it precise, for every $v \in \mathrm{Div}\,\mathcal{I}$ where $\mathrm{Div}\,\mathcal{I}$ is an intermediate subdivision toward the finest subdivision $\mathrm{Ch}^K\,\mathcal{I}$, let us define the *descendants* of v by $\pi^{-1}(v) = \{u \in \mathrm{Ch}^K\,\mathcal{I} \mid \pi(u) = v\}$, where $\pi : V(\mathrm{Ch}^K\,\mathcal{I}) \to V(\mathrm{Div}\,\mathcal{I})$ is the corresponding parent map. We present the optimized version of the generic protocol in Algorithm 4. (The omitted procedure WOScan is the same as in Algorithm 3.)

We notice that, for a particular decision map δ and each process i, the value $\delta(\pi^{-1}(v_i))$ can be precomputed for every possible vertex assigned to v_i in advance of actual execution of the protocol. This means that, specializing the code of Algorithm 4 w.r.t. the particular decision map δ, we can generate a further optimized implementation code.

Algorithm 5 gives such a code customized for the above renaming task for 3 processes. Observe that, precomputing descendant vertexes, program specialization has eliminated those redundant recursive calls which are not on reachable execution paths.

Here we notice that the optimization method discussed above is applicable to any protocol of the form $\delta \circ \mathrm{Ch}^K\,\mathcal{I}$. Furthermore, the optimized code is mechanically derived by specializing the generic protocol w.r.t. the concrete instance of decision map δ.

Although the program specialization gives a general optimization method, it heavily depends on each protocol instance how much memory access can be reduced. At one extreme, a protocol $(\mathcal{C}, \mathcal{C}, \pi \circ \mathrm{Ch}\,\mathcal{C})$, where $\mathcal{C} = \overline{\{v_0, \ldots, v_d\}}$ and $\pi : V(\mathrm{Ch}\,\mathcal{C}) \to V(\mathcal{C})$ is the parent map, is optimized to a protocol $(\mathcal{C}, \mathcal{C}, \Phi)$ with a trivial carrier map such that $\Phi(\sigma) = \overline{\sigma}$ that performs no shared memory access. At the other extreme, a protocol $(\mathcal{C}, \mathrm{Ch}\,\mathcal{C}, \iota \circ \mathrm{Ch}\,\mathcal{C})$ for chromatic agreement task, where $\iota : V(\mathrm{Ch}\,\mathcal{C}) \to V(\mathrm{Ch}\,\mathcal{C})$ is an identity vertex map, is not optimized at all by specialization,[2] since each vertex of $\mathrm{Ch}\,\mathcal{C}$ is assigned a different output. Thus, there is no general theorem on the reduction in the number or complexity of memory access. Furthermore, the code derived by the optimization method is, as is often the case

[2] To be precise, any protocol is specialized to a code that at least skips the subdivisions on 0-dimensional simplexes. The original implementation of immediate snapshot by Borowsky and Gafni can also be optimized likewise.

Algorithm 5 A customized code for the renaming task.

```
procedure IIS(k, d)                                   ▷ Code for process 0
    if v_0 = (0, {0,1,2}) then return 4
    else if d = 1 ∧ v_0 = 0 then return 0
    else if d = 1 then return 2
    else
        u ← WOScan(k, d)
        if u ≠ v_0 then v_0 ← u; IIS(k+1, n)
        else IIS(k, d−1)

procedure IIS(k, d)                                   ▷ Code for process 1
    if k = 1 ∧ d = 1 ∧ v_1 = 1 then return 0
    else if k = 1 ∧ v_1 = (1, {0,1}) then return 1
    else if k = 1 ∧ v_1 = (1, {1,2}) then return 2
    else if [ k = 2 ∧ v_1 = (1, τ) for some τ
            [ s.t. {(1, {0,1,2}), (2, {0,1,2})} ⊆ τ ] then return 2
    else if k = 2 ∧ d = 1 then return 3
    else
        u ← WOScan(k, d)
        if u ≠ v_1 then v_1 ← u; IIS(k+1, n)
        else IIS(k, d−1)

procedure IIS(k, d)                                   ▷ Code for process 2
    if k = 1 ∧ d = 1 ∧ v_2 = 2 then return 0
    else if k = 1 ∧ (v_2 = (2, {0,2}) ∨ v_2 = (2, {1,2})) then return 1
    else if [ k = 2 ∧ v_2 = (2, τ) for some τ
            [ s.t. {(1, {0,1,2}), (2, {0,1,2})} ⊆ τ ] then return 1
    else if k = 2 ∧ d = 1 then return 3
    else
        u ← WOScan(k, d)
        if u ≠ v_2 then v_2 ← u; IIS(k+1, n)
        else IIS(k, d−1)
```

with those obtained by automatic program generation, inevitably less structured than those protocols which are manually devised with human insights (e.g., the protocols [5, 10] for renaming task).

5 Conclusion and Future Work

We have shown that the multi-round protocol for the immediate snapshot by Borowsky and Gafni can be reformulated to conform to, in terms of combinatorial topology, Kozlov's construction of the standard chromatic subdivision via Schlegel diagrams. This gives a topologically smoother account for the protocol, where each round is simply a subdivision using Schlegel diagram. This topological simplicity has led to a straightforward method for optimizing distributed protocols defined by means of the iterated immediate snapshot: Each process executing the protocol narrows down the set of possible outputs per each round and can decide the final output at an earlier round, beyond which the same final output is reached no matter how the remaining rounds are executed.

The present paper exemplified that a topologically simpler modeling can better incorporate the theoretical results in topological studies on distributed computing into the more practical side of distributed systems. In this respect, it would be of interest of future investigation to generalize the result to encompass shared memory systems of different failure models such as [18, 9, 8]. Developing an appropriate multi-round protocol that operates on a topological model of a particular failure model, we would be able to optimize the corresponding class of protocols. Such an enhanced multi-round protocol would necessarily need to employ an augmented set of memory operations in a way that the topological structure induced from the extra operations (e.g., the one induced from the test-and-set operation [12]) is compatible with the failure model.

Acknowledgment. I would like to thank Nayuta Yanagisawa for his valuable comments on a draft of this paper. I would also like to thank the anonymous reviewers for their helpful comments.

References

1 Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002.
2 Fernando Benavides and Sergio Rajsbaum. The read/write protocol complex is collapsible. In *LATIN 2016: Theoretical Informatics - 12th Latin American Symposium*, volume 9644 of *LNCS*, pages 179–191. Springer, 2016.
3 Richard Bird. *Pearls of Functional Algorithm Design.* Cambridge University Press, 2010.
4 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing, STOC*, pages 91–100. ACM, 1993.
5 Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 41–51. ACM, 1993.
6 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations. In *Proc. of the 16th ACM Symposium on Principles of Distributed Computing*, pages 189–198, 1997.
7 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.
8 Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-write memory and k-set consensus as an affine task. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, pages 6:1–6:17, 2016.
9 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *ACM Symposium on Principles of Distributed Computing, PODC '14*, pages 222–231. ACM, 2014.
10 Eli Gafni and Sergio Rajsbaum. Recursion in distributed computing. In *Stabilization, Safety, and Security of Distributed Systems: 12th International Symposium, SSS 2010*, volume 6366 of *LNCS*, pages 362–376. Springer, 2010.
11 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.
12 Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects (preliminary version). In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 324–333. ACM, 1994.
13 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

14 Gunnar Hoest and Nir Shavit. Toward a topological characterization of asynchronous complexity. *SIAM J. Comput.*, 36(2):457–497, 2006.

15 Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall, 1993.

16 Dmitry Kozlov. *Combinatorial Algebraic Topology*. Springer, 2008.

17 Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(2):197–209, 2012.

18 Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *Distributed Computing - 30th International Symposium, DISC 2016*, pages 428–441, 2016.

19 Günter M. Ziegler. *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer, 1995.

A A Quick Look at Partial Evaluation

This appendix gives a brief overview of partial evaluation, a program optimization technique by program specialization. Partial evaluation is a matured field that has a long history of research. For details that cannot be covered in the following short overview, readers are advised to consult a textbook, say [15].

The fundamental idea of partial evaluation is quite simple. Suppose we are given a program and some of the expected inputs to it are known in advance. Then certain portions of the program may be *precomputed* w.r.t. the known inputs, by which the source program is transformed to an optimized one: The transformed program contains fewer computation steps to be performed at run-time. A subpart of the program is called *static*, if it does not depend on the inputs to be given at run-time; Otherwise, it is called *dynamic*. In particular, the known inputs are called static and the remaining inputs are called dynamic. It is the task of *binding-time analysis* to identify static parts as larger as possible for the chance of better optimization.

Let us see how a simple program that computes exponentiation n^m for non-negative integers n and m can be optimized by partial evaluation.[3] Such a program would be simply defined in a recursive style, as follows:

$$\mathrm{expt}(n, m) \;\equiv\; \mathsf{if}\ m = 0\ \mathsf{then\ return}\ 1\ \mathsf{else\ return}\ n \times \mathrm{expt}(n, m-1).$$

Suppose the second input m is known 9. Instantiating m with 9, we get:

$$\mathrm{expt}(n, \underline{9}) \;\equiv\; \mathsf{if}\ \underline{9 = 0}\ \mathsf{then\ return}\ \underline{1}\ \mathsf{else\ return}\ n \times \mathrm{expt}(n, \underline{9-1}),$$

where the underlined parts are the static ones, which are identified by binding-time analysis. Evaluating the static subexpressions, we obtain

$$\mathrm{expt}(n, \underline{9}) \;\equiv\; \mathsf{if}\ \underline{\mathit{false}}\ \mathsf{then\ return}\ \underline{1}\ \mathsf{else\ return}\ n \times \mathrm{expt}(n, \underline{8}).$$

Pruning the unreachable branch and unfolding the recursive call $\mathrm{expt}(n, \underline{8})$, we get

$$\mathrm{expt}(n, \underline{9}) \;\equiv\; n \times \big(\mathsf{if}\ \underline{8 = 0}\ \mathsf{then\ return}\ \underline{1}\ \mathsf{else\ return}\ n \times \mathrm{expt}(n, \underline{8-1})\big),$$

[3] This is the typical example that first appears in introductory texts of partial evaluation.

which reveals new static parts subject to further partial evaluation. Repeating this process, we will obtain the final transformation result:

$$\mathrm{expt}(n, \underline{9}) \equiv n \times n \times n \times n \times n \times n \times n \times n \times n.$$

Though the above simple example of partial evaluation improves the source program only marginally, just removing the overhead involved in conditional branching and recursive calls, the effect of optimization is amplified by applying it where execution bottleneck exists. In this paper, we are specifically concerned with application to the shared memory bottleneck. Another strength of partial evaluation is that the whole transformation process is mechanizable: Once we write a simple program, whose correctness is easier to reason about, we may automatically obtain one that is still correct yet optimized.

Optimization by partial evaluation, or program transformation by specialization in general, rarely improves computational complexity. In most cases, it improves efficiency only by a constant factor. As for the example of exponentiation, for instance, it is well known that an algorithm of logarithmic complexity is obtained by the technique of repeated squaring [7]. However, this kind of algorithmic leap usually needs human insights on the mathematical structure behind the problem to be solved. It is a central topic of program transformation how to optimize programs semi-automatically – mostly by mechanical 'calculation' on programs but with a little exploitation of the mathematical structure behind each particular problem. There has been lots of work done in this direction and several illuminating examples can be found in [3].

Designing a Planetary-Scale IMAP Service with Conflict-free Replicated Data Types

Tim Jungnickel[1], Lennart Oldenburg[2], and Matthias Loibl[3]

1 **Technische Universität Berlin, Germany**
`tim.jungnickel@tu-berlin.de`
2 **Technische Universität Berlin, Germany**
`l.oldenburg@mailbox.tu-berlin.de`
3 **Technische Universität Berlin, Germany**
`matthias.loibl@mailbox.tu-berlin.de`

Abstract

Modern geo-replicated software serving millions of users across the globe faces the consequences of the CAP dilemma, i.e., the inevitable conflicts that arise when multiple nodes accept writes on shared state. The underlying problem is commonly known as fault-tolerant multi-leader replication; actively researched in the distributed systems and database communities. As a more recent theoretical framework, Conflict-free Replicated Data Types (CRDTs) propose a solution to this problem by offering a set of always converging primitives. However, modeling non-trivial system state with CRDT primitives is a challenging and error-prone task. In this work, we propose a solution for a geo-replicated online service with fault-tolerant multi-leader replication based on CRDTs. We chose IMAP as use case due to its prevalence and simplicity. Therefore, we modeled an IMAP-CRDT and verified its correctness with the interactive theorem prover Isabelle/HOL. In order to bridge the gap between theory and practice, we implemented an open-source prototype *pluto* and an IMAP benchmark for write-intensive workloads. We evaluated our prototype against the standard IMAP server *Dovecot* on a multi-continent public cloud. The results expose the limitations of *Dovecot* with respect to response time performance and replication lag. Our prototype was able to leverage its conceptual advantages and outperformed *Dovecot*. We find that our approach is promising when facing the multitude of potential concurrency bugs in development of systems at planetary scale.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Geo-Replication, CRDT, Distributed Systems, IMAP, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.23

1 Introduction

When designing and developing a modern online service, researchers and developers are faced with a large and diverse spectrum of challenges. Due to its ever expanding reach, the Internet offers the unique while demanding prospect of connecting potentially billions of users scattered across the planet. The underlying distributed infrastructure requires enormous consideration on its own, but matters get even more involved when applications on top of it have state. While potentially possible, the penalties of declaring one cluster to be the single leader and routing all requests through it, render such a design unfeasible when reliability and responsiveness are of importance. A geographically-distributed system architecture, however, eliminates single points of failure and enables low response times on client requests.

Unfortunately, though, network and node failures are a given in large-scale infrastructures [2] and thus, our applications have to deal with partitions. To safeguard consistent state

during these, a trade-off between availability and consistency has to be made, commonly known as the CAP dilemma [11]. Choosing consistency requires minority partitions to reject client requests – becoming unavailable – while only a majority partition is allowed to make progress. This has been and still is a viable option, though with the rise of the NoSQL movement, available architectures have gained traction. They choose to always accept client requests even though all other nodes of the system might be temporarily unreachable and try to achieve consistency when connections are re-established. Naturally, in these *eventually consistent* [26] systems, conflicting requests arise. Much research has gone into efficient conflict resolution or even conflict avoidance.

More recently, *Conflict-free Replicated Data Types* (CRDTs) have been proposed as a method for avoiding conflicts [24]. If system state is built either on one of the state-based or operation-based data types, the inherent properties of these ensure that replicated state will always converge. Certain requirements have to be met in order to be able to use them, e.g., do we assume an underlying reliable causal-order network and commuting concurrent updates for the operation-based CRDTs. If we can provide these requirements, CRDTs are an elegant and efficient way to model available and partition-tolerant systems.

In this work, we set out to model, verify, implement, and evaluate a distributed application with non-trivial state based on CRDTs. As the provided service we chose to design an IMAP server. The *Internet Message Access Protocol* (IMAP) is the standard way to manage mailbox state and retrieve messages in an email service. IMAP is a simple and rather old standard – its beginnings date back to the mid-1980s – and as part of the email ecosystem is regularly proclaimed dead in favor of some supposedly more efficient communication service. Yet, email remains to be ubiquitous in all our lives and will stay so for the foreseeable future. As an example, Gmail recently crossed the mark of one billion monthly active users [19].

Even though the provided CRDT primitives [23] are concise and simple, one can fail in numerous ways when constructing non-trivial system state based on these. We wanted to be sure of the correctness of our model and thus put effort into proving it correct. To this end, we extended the CRDT and network model framework by Gomes et. al. [13, 12] written in the Isabelle/HOL interactive theorem prover to include our IMAP-CRDT. After being assured that state will always be consistent in our model, we adapted our prototype to adhere to the theoretical proof. This way, we achieve provable consistency in practice.

In order to evaluate the benefits of our proposed IMAP-CRDT, we developed a prototype *pluto* and built a federated Kubernetes[1]-based test environment on Google's public container cloud, Google Container Engine[2]. We were primarily interested in two characteristics: *response time performance* and *replication lag* [17]. The first one captures the perceived responsiveness by users while the second describes how long it takes one system to synchronize and apply updates with the other nodes. With our self-developed IMAP benchmark for write-intensive workloads, we were able to gather both insights for *pluto* and *Dovecot*.

We find, that our approach of designing distributed applications at planetary scale yields a straightforward flow of modeling, verifying, implementing, and evaluating software where a proven system model consolidates experimental measurements. The results attest the advantages of our CRDT-based architecture and draw near reproducibility due to our automated and open-sourced deployment infrastructure.

[1] `https://kubernetes.io`
[2] `https://cloud.google.com/container-engine`

Related Work. Large-scale distributed systems replicating application state in an available and partition-tolerant way have received academic attention since the advent of the Internet. Bayou [25] was one of the first distributed storage systems that enabled users to always submit updates and ensured eventual consistency when network connection was available again. Inspired by the fundamental concepts captured in Amazon's Dynamo paper [8], a new class of distributed data stores was proposed and developed, such as Cassandra [18] and Riak[3]. Many of these new developments are also based in parts on the ideas of Google's Bigtable concept [5], which Google itself turned into Spanner [6], its planet-scale strongly consistent and partition-tolerant distributed database. Their solution towards the CAP dilemma is to run Spanner on an expensive and highly sophisticated private network which ensures almost no downtimes [4].

Regarding automatic resolution of conflicting writes in any distributed system, the choice is between discarding all but one update or merging all updates into one. The most common technique for the first approach is known as *last write wins*, where the update with the biggest timestamp is picked as winner and all others are lost. One well-known merge-based resolution strategy is *operational transformation* [9], though mostly used for collaborative text editing and of decreasing performance with increasing number of operations [1].

Conflict-free Replicated Data Types take a different approach as they avoid conflicts altogether due to their construction properties. Apart from the set of basic data types defined in the original report [23], constructing further CRDTs has been an active research interest, for example, sequence CRDTs such as Treedoc [20] and LSEQ [22], and a composable JSON CRDT [16]. However, integration into production software is only progressing slowly, e.g., as part of Riak or AntidoteDB[4], and we don't see many approaches for standard IT services such as our IMAP-CRDT.

As part of this work, we verify the data type we propose within Isabelle/HOL. Formal verification of CRDTs has been done before, e.g., by Zeller et al. for state-based CRDTs [27]. The framework underlying our formal verification efforts was proposed and implemented by Gomes et al. [12], including a realistic network model and verification of the *Observed-Removed Set* (OR-Set). We base our IMAP-CRDT on the OR-Set and make the extensions to the framework by Gomes et al. with our data type accessible.

Considering state replication for the most widely used IMAP server, *Dovecot*, the *dsync*[5] approach is currently the only application-level support for planet-scale deployments. Unfortunately, *dsync* is limited to pair-wise replication, forfeiting the advantages of a cloud deployment with nodes all over the world. To our knowledge, we propose the first approach to a truly planetary-scale IMAP service that at the same time achieves low response times.

Finally, this work is based on previous efforts by us into the direction of an IMAP service based on CRDTs, presented in an earlier workshop paper [14].

Contributions. Our contributions in this work are as follows:

- We propose an IMAP-CRDT by modeling IMAP commands as operations on a CRDT. (Section 4)
- We verify the convergence of the IMAP-CRDT with Isabelle/HOL. (Section 4.1)
- We propose an open-source prototype *pluto* that offers IMAP at planetary scale with multi-leader replication based on CRDTs. (Section 5.1)

[3] `http://basho.com/products/riak-kv`
[4] `https://syncfree.github.io/antidote/`
[5] `https://wiki.dovecot.org/Replication`

- We introduce a benchmark for IMAP services. (Section 5.2)
- We propose a Kubernetes-based deployment for planet-scale *Dovecot*. (Section 6)
- We explore response time performance and replication lag of planetary-scale IMAP services on public clouds by evaluating the developed prototype *pluto* against state-of-the-art *Dovecot* setups. (Section 7)

2 System Model

In a traditional 3-tier system architecture of an IMAP service, a *proxy* (sometimes called *director*) forwards a client's request to a responsible *backend* node, where the request is processed. The service state in form of all users' mailboxes is typically stored on one or more *storage* nodes, traditionally hosting a strongly-consistent shared file system like NFS or GlusterFS. However, this architecture is no longer feasible in a planetary-scale IMAP system. To illustrate the pitfalls of this approach in a geo-replicated setting, we installed a GlusterFS on two virtual machines in different regions (North America and Europe) of the Google Cloud Platform (GCP). Subsequently, a *Dovecot* in the recommended setting with one proxy and three backend nodes was deployed on our Kubernetes cluster in Europe. Our benchmark revealed, that the response times of write requests exceed the round-trip time between North America and Europe by almost two orders of magnitude. Hence, applying a naive replication over two continents can be much more costly than routing all request through a single location.

The only solution to efficiently apply geo-replication is to relax the consistency requirements and allow backends to progress state without prior synchronization with other regions. This approach introduces the need of conflict management, because concurrent writes from multiple regions may be conflicting. The architecture underlying our proposed system enables a more fault-tolerant and responsive geo-replicated online service. In it, a backend node is authoritatively responsible for a particular range of users and asynchronously connected to backends in all other regions that are responsible for the same range of users. Furthermore, a global storage node can be added to the service that might run on single-tenant, high quality hardware for long-term storage. It can further be used as a failover destination if any backend node fails. From this *distributed system* of backends, we derive our system model which we use in Section 4.1 to reason about the convergence of state.

We obtain an asynchronous network of backend nodes which can be seen as independent processes. The network can suffer from partitions and can recover after a certain time. The backends continue to operate, even if the backend is temporarily disconnected from parts of the network. If a backend crashes, the state of the backend can be recovered. We assume non-byzantine behavior.

3 Technical Preliminaries

In this section we provide a brief introduction to CRDTs and IMAP. Both will be combined in Section 4 when we introduce our IMAP-CRDT.

3.1 Conflict-free Replicated Data Types

The theoretical concept of a Conflict-free Replicated Data Type has been formalized by Shapiro et al. in [24]. In essence, CRDTs enable convergence of replicas without requiring a central coordination server or even a distributed coordination system based on consensus or

locking. To achieve this goal, updates on application state based on CRDTs are designed to be conflict-free in the first place.

CRDTs come in two variants: *Convergent Replicated Data Types* (CvRDT) and *Commutative Replicated Data Types* (CmRDT). CvRDTs, often described as state-based CRDTs, ensure convergence by defining a *merge* function that is applied on two diverged states in order to obtain a consistent state again. The merge function calculates the *least upper bound* on a *join semi-lattice*, and therefore must be commutative, idempotent, and associative. A replica can update its local state and send the updated version to all other replicas which individually apply the merge function to regain a consistent state. The order in which the merge function is applied, is irrelevant.

In this work we focus on the operation-based variant, the CmRDT. In contrast to state-based ones, replicas in this case exchange operations directly, with minimal state information. A reliable causal-order broadcast ensures that operations ordered by *happened-before* relation on the source replica are received and applied accordingly at all other replicas. Updates that cannot be ordered by *happened-before* are considered *concurrent* and required to commute. The design of a CmRDT is a challenging task, fortunately the technical report offers a variety of specifications for counters, sets, graphs, and even lists [23].

As mentioned, CmRDTs require a reliable causal-order broadcast to ensure causal consistency. Note, that an implementation of such a broadcast does not require consensus and can be achieved by use of vector clocks.

With commutativity of concurrent updates and reliable causal-order broadcast, Shapiro et al. showed that any two replicas that have seen the same set of operations have equivalent abstract states and therefore eventually converge.

3.2 IMAP

An IMAP service manages mailboxes of registered users. Users are able to interact with their mailboxes by sending IMAP commands to the server. These commands are defined in the *IMAP4rev1* standard in RFC 3501 [7]. To reduce the complexity of this paper, we focus on the *consistency-critical* commands, i.e., commands that change the server's state. The commands *create* and *delete* respectively add or remove a mailbox (also called mailbox folder, or simply folder) to or from a user's account. An *append* command is used to add a message to a mailbox folder. With *store*, the message flags, e.g., `Seen`, `Answered`, or `Deleted`, can be altered. The *expunge* command is used to remove all messages with such a `Deleted` flag from a particular mailbox folder. Note, that the commands *store* and *expunge* are only allowed once a particular mailbox folder has been selected with the *select* command.

IMAP servers, like *Dovecot*, support various formats to represent the mailboxes of the users on hard disk. Typical formats are *mbox* and *Maildir*. The latter is generally preferred due to its use of individual files per mail message and thus, no locking is required when messages are appended. The messages are given unique file system names that include any potential standard flag.

4 IMAP-CRDT

In our scenario, the main challenge is to model the IMAP commands as operations on a CmRDT. We begin with the decision on the used payload, i.e., the underlying state representation. We identified a map that projects folder names to the content of a folder to be best suited. Therefore, the content of a folder is a combination of metadata (tags) and messages. We model the map as a function $u : \mathcal{N} \to \mathcal{P}(\texttt{ID}) \times \mathcal{P}(\mathcal{M})$ where $\mathcal{N}$ is the set of

Specification 1 IMAP-CRDT (payload, *create*, and *delete*).

```
1: payload map u : 𝒩 → 𝒫(ID) × 𝒫(ℳ)      ▷ {foldername f ↦ ({tag t}, {msg m}), ...}
2:     initial (λx.(∅, ∅))
3: update create (foldername f)
4:     atSource
5:         let α = unique()
6:     downstream (f, α)
7:         u(f) ↦ (u(f)_1 ∪ {α}, u(f)_2)
8: update delete (foldername f)
9:     atSource (f)
10:        let R_1 = u(f)_1
11:        let R_2 = u(f)_2
12:    downstream (f, R_1, R_2)
13:        u(f) ↦ (u(f)_1 \ R_1, u(f)_2 \ R_2)
```

foldernames, ID is the set of tags, and $\mathcal{M}$ is a set of messages. We denote $\mathcal{P}(X)$ to be the power set of X.

Because a folder f contains arbitrary items, the result of $u(f)$ is a tuple of two sets. The first set, denoted as $u(f)_1$, is the set of tags that represent metadata that should not be visible to a user. The second set, denoted as $u(f)_2$, represents the messages in the folder.

If both sets $u(f)_1$ and $u(f)_2$ are empty, the folder is interpreted as non-existent. Note, that we distinguish between a non-existent folder and an empty folder. A folder is empty, if $u(f)_2$ is empty but $u(f)_1$ is not empty, i.e., certain metadata is present. Initially, all folders are non-existent. Hence, the initial state can be described as a lambda abstraction that projects the tuple $(\varnothing, \varnothing)$ to every folder name in $\mathcal{N}$.

We present the complete IMAP-CRDT in Spec. 1 and Spec. 2. We adhere to the presentation style that has been introduced by Shapiro et al. in [24]. Next, we define the operations that represent the IMAP commands and begin with *create* and *delete* in Spec. 1.

The desired result of *create* is to create an empty folder f. Therefore, a fresh and unique tag t is generated on the replica that initiates the operation. This initiation phase of the replica is usually called **atSource**. Thereafter, the tag t is inserted into $u(f)_1$ and $u(f)_2$ remains untouched. This part of the operation is usually called **downstream** and is executed at every replica. We denote an update of the map entry as $u(f) \mapsto (X, Y)$ where X and Y are the new sets that override the existing sets. Note, that the map entries for the other folder names remain unchanged.

In contrast to *create*, the desired result of the *delete* operation is to make the folder non-existent. Hence, the content of $u(f)$ is removed at every replica. If we defined the downstream operation to be $u(f) \mapsto (\varnothing, \varnothing)$, then *create* and *delete* would no longer be commutative. Furthermore, the IMAP specification requires any *delete*(f) to be preceded by a *create*(f), aborting on IMAP protocol level if a client tries to remove a non-existing folder. This eliminates consistency issues when *delete*(f) and *create*(f) are issued concurrently. Note, that the definitions of *create* and *delete* are very similar to the *add* and *remove* operations on the op-based Observed-Remove-Set (OR-set), which has been introduced in [24].

The remaining operations *append*, *expunge* and *store* are defined in Spec. 2. The *append* operation is very similar to the *create* operation, except that a message m is inserted into $u(f)_2$ and $u(f)_1$ remains unchanged. Another important difference is the atSource precondition. The IMAP specification states, that each message is assigned a unique identifier called

Specification 2 IMAP-CRDT (*append*, *expunge*, and *store*).

14: **update** *append* (foldername f, message m)
15: **atSource** (m)
16: **pre** m is globally unique
17: **downstream** (f, m)
18: $u(f) \mapsto (u(f)_1, u(f)_2 \cup \{m\})$
19: **update** *expunge* (foldername f, message m)
20: **atSource** (f, m)
21: **pre** $m \in u(f)_2$
22: let $\alpha = \mathit{unique}()$
23: **downstream** (f, m, α)
24: $u(f) \mapsto (u(f)_1 \cup \{\alpha\}, u(f)_2 \setminus \{m\})$
25: **update** *store* (foldername f, message m_{old}, message m_{new})
26: **atSource** (f, m_{old}, m_{new})
27: **pre** $m_{\text{old}} \in u(f)_2$
28: **pre** m_{new} is globally unique
29: **downstream** $(f, m_{\text{old}}, m_{\text{new}})$
30: $u(f) \mapsto (u(f)_1, (u(f)_2 \setminus \{m_{\text{old}}\}) \cup \{m_{\text{new}}\})$

`UID`. We use this requirement to assure that no two *identical* messages are ever appended by different replicas, or even the same replica. Note, that *identical* is not referring to the message content. In practice, it is still possible to append two messages with identical content, although the `UID`s of the messages are in fact different.

The operation *store* is implemented in a similar fashion. The main purpose of *store* is to change the flags of a message m_{old}. We do not explicitly model the flags of a message. Instead, we insert the message m_{old} with updated flags as a new message m_{new} in $u(f)_2$ after deleting m_{old} from $u(f)_2$.

In contrast to the previous definitions, the *expunge* operation is rather counter-intuitive. The deletion of a message, which has been marked with a `Deleted` flag, is simply done by removing the message from $u(f)_2$. However, we decided that an additional tag must be inserted into $u(f)_1$ to avoid unexpected behavior in combination with a concurrent *delete* operation. We illustrate this puzzle with the following example.

Two replicas r_1 and r_2 initially share the following state of a folder: $u(f) = (\varnothing, \{m_1\})$. The replica r_1 initiates a *delete* operation, resulting in an update of the local state at r_1 to be $u(f) = (\varnothing, \varnothing)$ and f is interpreted as non-existent, i.e., the complete folder is deleted. In the meantime, r_2 independently initiates an *expunge* operation that aims to delete m_1, resulting in the local state to be $u(f) = (\{t_{42}\}, \varnothing)$, i.e., an empty folder. At this point, it is unclear what result is actually desired, after the downstream operations are executed at both replicas. We decided, that the folder should be present as an empty folder at both replicas. Hence, according to the presented definitions the resulting state is $u(f) = (\{t_{42}\}, \varnothing)$. In fact, our definition of the operations gives *create*, *append*, *store*, and *expunge* a precedence over *delete*, i.e., when manipulations of the folder f and a *delete*(f) are concurrently executed, the folder is never entirely deleted, only state visible at the initiation time of the *delete* operation is removed. Hence, we decided to pursue an *add-wins* semantic.

Design Decisions and Discussion. The proposed *add-wins* strategy comes at the price of increased metadata that needs to be managed. In the presented definition, we create a new tag for each deleted message of an *expunge* operation. These tags are currently only removed

by a *delete* operation, which is typically not executed as long as the user holds interest in the folder. To overcome this issue, some metadata could be deleted after a certain *stable* state has been reached. For example, Baquero et al. introduced the notion of log compaction through *causal stability information* in [3]. An alternative decision would be to give *delete* precedence over the other operations. In this case, less metadata would be required to process state information. However, the application behavior in the presence of concurrent updates seems undesired. For example, in case of a concurrent *append* and *delete* operation on the same folder, the message that was added by the *append* operation would be deleted with the folder and be lost forever. Note, that our IMAP-CRDT requires causal-order delivery and we omit this precondition in every update operation for the sake of simplicity.

The commands left to achieve full compliance with RFC 3501 are mostly *read* commands like *search*, *fetch*, or *status*, and thus not in the scope of this work. Missing *write* commands like *copy* and *rename* bear many parts from the already implemented *write* commands but require further careful thought. However, modeling those commands as operations on the IMAP-CRDT is an effortful but realizable task.

4.1 Verification

To ensure that all required properties are satisfied and that convergence among replicas is achieved, we verified our IMAP-CRDT with the interactive theorem prover Isabelle/HOL. We base our Isabelle/HOL implementation on the recently published CRDT verification framework by Gomes et al. [12]. Our formalization follows the definitions we presented in Spec. 1 and Spec. 2 and is available in the *Archive of Formal Proofs* for Isabelle/HOL [15]. The only notable difference in the Isabelle/HOL formalization is, that we no longer distinguish between sets `ID` and $\mathcal{M}$ and that the generated tags of *create* and *expunge* are handled explicitly. This makes the formalization slightly easier, because less type variables are introduced. Ultimately, we show that our IMAP-CRDT achieves *strong eventual consistency* in our proposed system model, i.e., all replicas converge to an identical abstract state when they share the same history of operations.

5 Prototype

5.1 Pluto

Our prototypical implementation of the presented IMAP server system model, *pluto*, and all other software we wrote for this paper, is developed in the Go programming language and available[6] as open-source software under GPLv3 license. The prototype is designed as a distributed IMAP server, internally comprised of multiple *distributor* nodes, multiple *worker* (backend) nodes, and one *storage* node, as set out in Section 2. We put emphasis on application speed, security, reliability, and configurability.

In any *pluto* deployment, an IMAP request enters the service at a stateless distributor node. Any request initiated via an unencrypted connection will get dropped, ensuring that authentication credentials transmitted as part of an ordinary IMAP session are only ever sent over a TLS connection. The distributor node handles a session as far as the authentication procedure was successful. For any further request, the worker node responsible for the partition of users the current user is part of, is determined, and all traffic is proxied to this

[6] `https://github.com/go-pluto/pluto`

node via a gRPC[7] connection. Should the determined worker node be unavailable due to any number of reasons, a *failover* to the global storage node is performed, which accepts the proxied IMAP traffic in place of the worker node.

As soon as requests of a regular IMAP session reach a worker or storage node, they potentially change the mailbox state of the respective user. To achieve availability even in case of failures, worker and storage nodes accept these state-changing requests and guarantee that eventual consistency with the other replicas is reached – an inherent feature of CRDTs. We say that worker and storage nodes are stateful because they first alter their local states and afterwards send messages downstream that apply the same operation on all remote states. This makes *pluto* a multi-leader replication system.

We verified the correctness of our state replication as part of Section 4 and implemented the two required components, the IMAP-CRDT and reliable causal-order broadcast of update messages, as parts of *pluto*. For the IMAP-CRDT, we assign each user an OR-Set, called *structure*, that represents the user's abstract mailbox state. The main difference to our theoretical model in Spec. 1 is, that the map $u(f)$ for mailbox folder f is modeled as a set of *value-tag* pairs for which the *value* element is always set to f. As an example, we consider a mailbox folder `uni`, on which an *append* operation was executed. Assume, that the state according to Spec. 1 looks like $u(\texttt{uni}) \mapsto (\{\alpha\}, \{m\})$. We can infer that the *create* operation for `uni` created tag α in $u(\texttt{uni})_1$ and the *append* operation put m into $u(\texttt{uni})_2$. In our *structure* OR-Set this is represented as $\{(\texttt{uni}, \alpha), (\texttt{uni}, m)\}$. Thus, in *pluto* we do not distinguish between metadata and message tags. Any update to *structure* is followed by a file system `sync` operation on an associated log file on stable storage. This ensures that nodes can precisely reconstruct the internal representation of user mailboxes in case they crash.

An update on a source replica triggers a message to all downstream replicas in order to reproduce it on their state. In *pluto*, worker and storage nodes are grouped into subnets that exchange updates for a particular partition of users. Considering a planetary-scale deployment with workers in Europe, the US, and Asia, and the storage in Australia, the subnet for a worker `eu1` in Europe might contain `us1`, `asia1`, and `storage`. Each downstream message from `eu1` is sent to all other nodes from its subnet. As the IMAP-CRDT is based on the operation-based OR-Set, we require these messages to be part of a reliable causal-order broadcast, ensuring that they are delivered to the application exactly once and with no causally-preceding ones missing. To this end, we maintain vector clocks [21, 10] for each subnet. Send queue, receive queue, and vector clock are again `sync`'ed into associated files on any update. To reduce replication lag, we do not send messages individually but transfer the current send log as a whole in a defined interval.

IMAP clients will not notice the replication efforts, as they happen asynchronously. This is one of the major advantages of a geographically distributed *pluto* deployment: requests can be responded to quickly due to authoritative local state on close-by worker and storage nodes while updates will get applied everywhere eventually due to CRDTs and reliable causal-order broadcast, thus achieving consistent state. We guide these claims with structured logging, metrics exposure to Prometheus, and tests for important packages. Further work might go into file checksum checking for ascertaining data integrity, deeper performance profiling, and increased RFC 3501 [7] compliance. We welcome contributions from the community.

[7] `https://grpc.io`

5.2 Benchmark

In order to evaluate *pluto* and *Dovecot* in Section 7, we needed a way to apply a large amount of state-changing IMAP commands to our deployments. We are interested in the state-changing ("write") commands of RFC 3501 because only these manipulate mailbox state and trigger downstream messages that need to be applied at other replicas. "Read" commands in turn are answered authoritatively on the replica they are received on, without replica communication. Only state-changing commands potentially unearth consistency issues by generating edge cases. Thus, we required an IMAP benchmark that is able to generate large write-intensive workloads involving the write commands that are implemented in both services: *create*, *delete*, *append*, *expunge*, and *store*.

We could not find such tool or data set available, and thus implemented an IMAP benchmark ourselves[8] that generates arbitrary amounts of random data, write-intensive workloads. Each workload is composed of small and randomly generated sequences of IMAP commands, called *sessions*. One session always contains well-matched IMAP commands, e.g., a mailbox folder is created before a message is appended to it. Session generation is deterministic and can be reproduced by configuring a benchmark with the same seed.

Before a session is executed, a user is chosen randomly from a provided users file and logged in. Next, the session commands are applied one after another, a successive one as soon as the current one has finished and the time between sending it and receiving a complete answer – the command's *response time* – has been stored. The workload's degree of parallelism, that is, the number of concurrent active users, can be configured as well. The results are written to disk and optionally uploaded to a Google Cloud Storage (GCS) bucket.

5.3 Maildir Tools

With the benchmark ready we had almost everything in place required for putting our system model to a test. One more component was needed, though, for gaining insight into the replication performance. While the IMAP benchmark provides response time measurements, replication lag data is at least as important because it tells us how well a service is able to disseminate and apply updates among its replicas. It complements the user-centric response time metrics by making visible the asynchronous replication part. Due to different replication mechanisms in *pluto* and *Dovecot*, though, we had to fall back to observing the Maildir file system in order to see when updates were applied.

We implemented a small utility[9] that periodically performs a disk usage calculation of a configured subset of the Maildirs present on a node (by running `'du -s'`). The results are logged to disk and uploaded to a GCS bucket at the end of the tool's run. For continuous monitoring, a duration histogram is exposed to Prometheus. The idea is to integrate one Maildir dumper into each stateful node deployed in a service to be evaluated. After having run an IMAP benchmark, timestamped disk usage reports can be collected and the time difference between the points in time when two observed Maildirs report the same size in bytes can be calculated. We consider this measure the replication lag. Please note, that the calculated time differences have to be taken as estimations rather than precise durations as we rely on synchronized clocks for timestamp elicitation. In Section 7, we will see that the clock synchronization in Google's data centers has negligible influence on our results.

[8] `https://github.com/go-pluto/benchmark`
[9] `https://github.com/go-pluto/maildir_tools`

6 Infrastructure

We now introduce the infrastructure setup used in our experiments later on. As guiding principle, we have chosen a *Cloud Native* approach, featuring the most advanced cloud technologies available at the time of developing our prototype. Our infrastructure is mainly based on two products: Kubernetes, an orchestration platform for containerized applications, and Prometheus, a powerful monitoring tool.

We provisioned two identical Kubernetes clusters in the `us-east1-b` and `europe-west1-b` regions of the Google Cloud Platform. Each cluster consisted of six `n1-standard` nodes (1 vCPU, 3.75GB memory). We combined both clusters into a Kubernetes cloud federation, enabling cross-cluster service discovery and resource synchronization. For persisting data, we always allocated 100GB SSD volumes. In the following, we will write `us` or `europe` in reference to the respective regions.

We decided to publish our configurations in our infrastructure repository[10], so that our setup can easily be re-created and re-used for further experiments. Hence, all resources, including the container images of all evaluated systems, are publicly available.

7 Evaluation

To evaluate our approach, we conducted a set of experiments. We started by defining our *baseline*, i.e., a reference experiment where we used a standard configuration of *Dovecot* without any replication. Thereafter, we conducted two experiments where we compared *pluto* against *Dovecot* with enabled replication. The results of these experiments we will discuss in the end of this section.

7.1 Baseline Experiment

As introduced in Section 2, we used a *Dovecot* in a traditional 3-tier architecture as reference setup. For the storage layer, we deployed a *GlusterFS* with a replicated volume on two `n1-standard` nodes with 100GB SSDs in the `europe` region. The remaining *Dovecot* components, i.e., a proxy and three backends, were installed on our Kubernetes cluster in the same region as *GlusterFS*. We used three backend nodes to illustrate the possibility of partitioning (also known as *sharding*). In this and all later experiments we maintained a total number of 120 active users in three static user partitions and the proxy was configured to redirect users to the backend that was responsible for their partition. In this setup, no replication was introduced besides the synchronized volume in the *GlusterFS* cluster.

We configured our IMAP benchmark (see Section 5.2) to execute 5000 IMAP sessions with a session length between 15 and 40 commands. The degree of parallelism, i.e., the number of users that are concurrently executing sessions, was set to 20. These 20 concurrent users were identified to be best suited for our experiments, because the reference setup reached the best resource utilization at reasonable response times.

We executed our benchmark on our Kubernetes cluster in the `us` region to simulate a write-intensive load from a distant location. In other words, we used a workload that required geo-replication on a system that was not replicated. Thus, high response times were expected but no replication lag.

[10] `https://github.com/go-pluto/infrastructure`

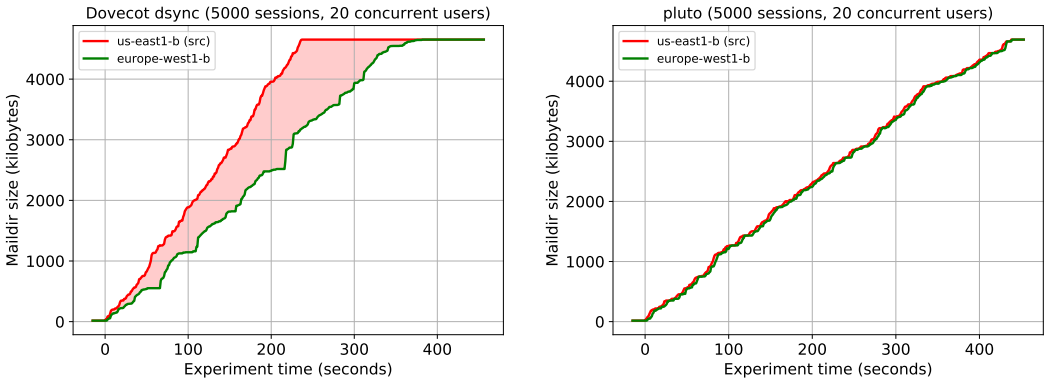


Figure 1 Replication lag diagram for *Dovecot dsync* (left) and our prototype *pluto* (right) for requests from `us` to `europe`.

We call this experiment our *baseline*, because all geo-replicated setups must be able to outperform it. Otherwise, the effort of geo-replication and the introduction of a replication lag is pointless.

Results. We show the measured response times in the *baseline* column of Table 1. The average and median response times in milliseconds are grouped by IMAP command. We judge the measured values as realistic for this setup. In fact, our findings in [14] confirm the authenticity of the presented values.

7.2 Experiment 1: Single-Cluster Benchmark

In the remaining experiments, we focused on the systems that offer multi-leader replication, namely *dsync* and *pluto*. We deployed a setup of one proxy (or director) and three backends (or workers) in the `europe` and `us` Kubernetes clusters. Both setups were connected over a Kubernetes federation and communicated over public IP addresses and TLS-encrypted channels. In the first experiments, we replayed the settings from our *baseline* experiment, except that the traffic from the `us` region was now directed to the respective proxy in the same region. In this scenario, the expected behavior is that both systems replicate the updated application state from `us` to `europe` asynchronously. During the run, we collected the response times and additionally tracked the size of the mailboxes for six selected sample users in both regions with our Maildir tools (see Section 5.3). The tracking interval was set to one second, which we found to be the best trade-off between additional overhead by the `du` commands and unavoidable loss of precision. With the chosen interval, a possible *micro clock drift* between `europe` and `us` has no significant influence to our results. Based on the collected values, we identified the replication lag for both systems. We compare the results for *dsync* and *pluto* in the following two paragraphs.

Results: Dovecot dsync. The measured response times are given in the *dsync* column of Table 1. We judge the response times and the resulting throughput, i.e., the processed IMAP commands per second, as optimal for this setup. *Dovecot* is – not for nothing – the state-of-the-art IMAP server software.

For analysis of the replication lag, we compare the growth of the mailboxes in both regions for the selected sample users. In the left side of Figure 1, we illustrate the average

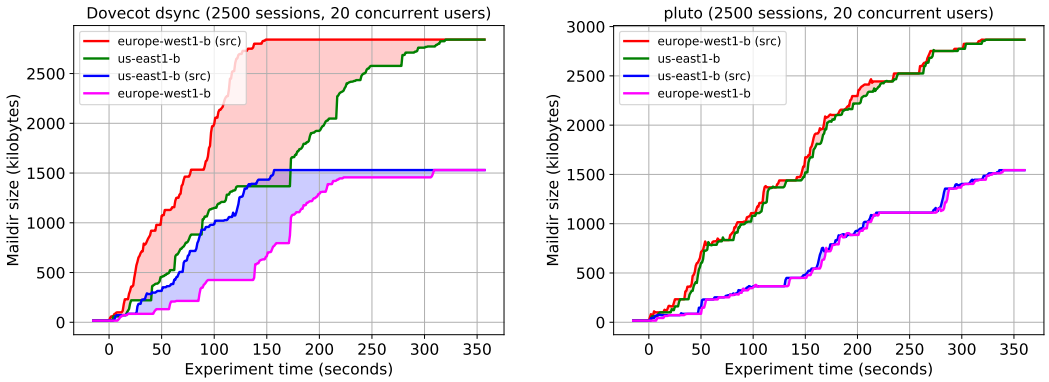


Figure 2 Replication lag for *dsync* (left) and *pluto* (right). The red areas represent the replication from `europe` to `us` while the blue areas represent the opposite direction.

growth for the selected sample users in what we call a *replication lag diagram*. On the x-axis we see the relative time of the experiment in seconds. The y-axis represents the size of the users' mailboxes in kilobytes. The red line represents the growth of the mailboxes in `us`, i.e., the region where the traffic was injected. The green line represents the growth of the replicated mailboxes in `europe`. In this replication lag diagram, a distance between both curves parallel to the x-axis represents the replication lag in seconds, i.e., the time until the `europe` replica catches up. A distance between both curves parallel to the y-axis represents the replication lag in kilobytes[11]. In order to quantify the replication lag, we think that it is feasible to compute the size of the red area between both curves. The computed area in *megabyte*second*, alongside with the average and median replication lag in kilobytes, is presented in the last 3 rows of Table 1.

Results: pluto. For the *pluto* setup, we additionally deployed the *storage* node (see Section 5.1) in a third region (`europe-west2-b`). Because we cannot directly compare the storage node to any *Dovecot* component, we used a more powerful node (`n1-standard-4`, 4vCPU, 15GB Memory) and set the resolution of our Maildir tool for this node to 3 seconds to avoid any negative impact. The remaining parts of the *pluto* setup is almost identical to *dsync*, i.e., we have one director and three worker nodes with 100GB SSDs in each region.

The measured response times are stated in the *pluto* column of Table 1. We note that the response times are significantly higher than *Dovecot*'s, which we discuss in the end of this section.

The replication lag diagram is shown in the right part of Figure 1. We see that the difference between the curves is almost invisible, which indicates a very small replication lag. The quantified replication lag is shown in Table 1.

7.3 Experiment 2: Double-Cluster Benchmark

For our final experiment, we split the workload from the previous experiments and used our benchmark from both regions `us` and `europe`, i.e., we executed 2500 sessions from each region to simulate a workload that, in fact, requires geo-replication. The measured response times are stated in the $dsync^2$ and $pluto^2$ columns of Table 1.

[11] We note, that these diagrams require a monotone growth of the mailboxes to be meaningful. Our benchmark generates *mostly* monotone growth, because *create* and *append* commands are more likely than *delete*. With the chosen resolution of our Maildir tools, a declining mailbox size is almost invisible.

Table 1 The combined results of Experiment 1 and 2, showing the response time performance in milliseconds and the throughput in IMAP commands per second. The average and median replication lag is stated in kilobytes and the replication lag area is stated in megabyte*second.

			baseline	*dsync*	*pluto*	*dsync*^{2}		*pluto*^{2}	
						`us`	`eu`	`us`	`eu`
Response Time Performance	`CREATE`	Average	251.36	16.24	47.77	18.47	23.24	47.56	75.20
		Median	224.50	12.52	28.25	14.17	20.33	28.94	29.83
	`DELETE`	Average	602.05	30.81	48.85	32.46	37.03	47.12	74.61
		Median	539.38	27.84	28.30	29.46	34.31	29.16	29.89
	`APPEND`	Average	437.26	43.02	91.96	46.39	55.36	87.23	131.79
		Median	400.87	38.37	57.15	42.08	50.34	55.72	58.66
	`EXPUNGE`	Average	112.91	13.87	42.74	15.59	21.16	40.94	62.72
		Median	97.05	**6.18**	25.61	**9.44**	**18.91**	22.56	23.19
	`STORE`	Average	184.16	15.72	52.04	17.48	21.79	46.09	72.84
		Median	166.66	**11.93**	31.80	**13.83**	**19.64**	29.73	31.53
	Throughput		47.17	480.67	256.03	447.87	367.94	256.26	171.49
Replication Lag		Average		734.61	39.10	592.87	657.76	18.61	44.98
		Median		729.10	**34.44**	217.83	322.10	**6.1**	**34.33**
		Area		279.89	**17.13**	97.92	209.83	**5.83**	**14.32**

In order to measure the replication lag, we also split the sample users and configured our benchmark in a way that the mailboxes of the first half of the users are only accessed by the `us` benchmark, and the second half by the `europe` benchmark. The mailboxes of the remaining 114 users receive commands from both regions. We present the replication lag diagram for both systems in Figure 2. The red areas represent the replication lag for synchronizing state from `europe` to `us`, and the blue areas represent the replication lag in the opposite direction.

7.4 Discussion

The *baseline* experiment revealed that the absence of geo-replication can be costly with respect to response time and throughput, when the application is faced with traffic from distant regions. As we have seen with both compared systems, using multi-leader replication for traffic from different continents is convincing and necessary. The price for the introduced replication is relaxation of consistency guarantees and presence of a replication lag.

By comparing the response times of both systems, and in extension to that, the achieved throughput, we clearly see that our prototype cannot keep up with *Dovecot* and that further optimizations are necessary. We acknowledge, that throughput often is a performance metric that is placed emphasis on in large-scale services and *pluto* needs to improve in that direction. However, because *pluto* is a research prototype with much less development time compared to the standard IMAP server *Dovecot*, we nevertheless are satisfied with its response time performance. We think, that optimizations of the used index structures and file management can lead to improved response times and throughput.

With respect to the replication lag, our prototype clearly outperforms *dsync* and we judge our approach as successful. Replication based on the used op-based CRDT is cheap compared to the costly replication of *dsync*. An operation from one replica can almost instantly be delivered and applied on the other replicas without complex tracking of state information. The fact that our approach can be applied with an arbitrary number of replicas makes it even more interesting than *dsync*, where only a pair-wise replication is possible.

We note, that our experiments only focus on write-intensive workloads and we purposely omitted the evaluation of read commands. Building an IMAP server that is able to compete with *Dovecot* in all facets is a challenging task, and is, at least for now, not our primary focus. In our opinion, the improvement of our IMAP-CRDT and exploration of further standard IT services that can be modeled with CRDTs, is a promising direction for future work.

We would like to point out, that we chose IMAP as the protocol to model with a custom CRDT not because it is better suited for this purpose than other protocols. We chose IMAP because of its widespread use and fundamental importance in everyday life – and, because its relative simplicity allowed for completing work on time. We judge the fact that application state of an IMAP server is based on relatively simple structures, namely its tree-like mailbox structure, as particularly advantageous for modeling the commands with operations on a CRDT. Hence, as long as the structural complexity of application state to model is manageable, our approach is promising. We expect, that with the recently introduced JSON CRDT [16], the modeling of more IT services with CRDTs will become even easier. However, a machine-checked verification of the JSON CRDT is still to be done.

8 Conclusion

The initial exploration of the feasibility of using CRDTs in the multi-leader replication of an IMAP service can be considered successful. We have made two important contributions: a verified IMAP-CRDT design and the evaluation of our prototype, where we showed that the replication lag can be significantly reduced compared to *dsync*, the replication tool of the de-facto standard IMAP server *Dovecot*.

In our work, we consider IMAP as the example to show the benefits of modeling standard IT services with CRDTs. Offering multi-leader replication without the need of manual conflict resolution enables not only the possibility of planet-scale distributed applications, but also more reliability in the presence of failures. To emphasize this further, this work convinced us that really any stateful IT service should be examined for applicability of multi-leader replication. Relying on strongly-consistent operations and fault-free infrastructure can get risky as state becomes ever more shared and clients distributed. CRDTs combined with formal verification offer the means to achieve confidence in relaxed consistency. Thus, considering this approach when designing and even upgrading large-scale IT services can be a matter of securing viability of a particular service – even in a single data center deployment.

Our approach, where we began with the system design and verification followed by the implementation and evaluation, turned out to be successful in this regard. The resulting prototype combines *theory* and *practice* by leveraging CRDTs in a standard IT service and is able to play off its conceptual advantages. We encourage fellow system designers to follow in this path and consider CRDTs for modeling application state.

Future work includes the exploration of CRDTs for other everyday IT services and further improvement of our prototype.

Acknowledgements. We would like to thank the Software Technology Group at TU Kaiserslautern, Georges Younes, Vitor Enes, and the anonymous reviewers for their valuable comments and feedback. Furthermore, we thank the German Research Foundation (DFG) and the graduate school SOAMED for supporting this work.

References

1. Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In *ACM Symposium on Document Engineering*, DocEng'11, pages 103–112, 2011.
2. Peter Bailis and Kyle Kingsbury. The Network is Reliable. *Commun. ACM*, 57(9):48–55, 2014.
3. Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems*, DAIS'14, pages 126–140, 2014.
4. Eric Brewer. Spanner, TrueTime and the CAP Theorem. Technical report, Google, 2017.
5. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 15–15, 2006.
6. James C. Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Google's Globally-distributed Database. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.
7. Mark R. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501, University of Washington, 2003. URL: `https://www.rfc-editor.org/rfc/rfc3501.txt`.
8. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles*, SOSP'07, pages 205–220, 2007.
9. C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Rec.*, 18(2):399–407, 1989.
10. Colin J. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*. Australian National University. Department of Computer Science, 1987.
11. Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.
12. Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes. *Archive of Formal Proofs*, 2017. `http://isa-afp.org/entries/CRDT.html`.
13. Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.*, 1(109):1–28, 2017.
14. Tim Jungnickel and Lennart Oldenburg. Pluto: The CRDT-Driven IMAP Server. In *International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC'17, pages 1–5, 2017.
15. Tim Jungnickel, Lennart Oldenburg, and Matthias Loibl. The IMAP CmRDT. *Archive of Formal Proofs*, 2017. `http://isa-afp.org/entries/IMAP-CRDT.html`.
16. M. Kleppmann and A. R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.
17. Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., 2017.
18. Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.
19. Frederic Lardinois. Gmail Now Has More Than 1B Monthly Active Users. *techcrunch.com*, 2016. [Online; posted February 1, 2016]. URL: `https://tcrn.ch/1nJbAAe`.

20 Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. *Operating Systems Review*, 44(2):29–34, 2010. doi:10.1145/1773912.1773921.

21 Friedemann Mattern. Virtual Time and Global States of Distributed Systems. *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

22 Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *ACM Symposium on Document Engineering*, DocEng'13, pages 37–46, 2013.

23 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical report, Inria, 2011. URL: https://hal.inria.fr/inria-00555588/file/techreport.pdf.

24 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, 2011.

25 D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *ACM Symposium on Operating Systems Principles*, SOSP'95, pages 172–182, 1995.

26 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, 2009.

27 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal Specification and Verification of CRDTs. In *Formal Techniques for Distributed Objects*, FORTE'14, pages 33–48, 2014.

Non-Uniform Replication*

Gonçalo Cabrita[1] and Nuno Preguiça[2]

1 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
g.cabrita@campus.fct.unl.pt
2 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
nuno.preguica@fct.unl.pt

Abstract

Replication is a key technique in the design of efficient and reliable distributed systems. As information grows, it becomes difficult or even impossible to store all information at every replica. A common approach to deal with this problem is to rely on partial replication, where each replica maintains only a part of the total system information. As a consequence, a remote replica might need to be contacted for computing the reply to some given query, which leads to high latency costs particularly in geo-replicated settings. In this work, we introduce the concept of non-uniform replication, where each replica stores only part of the information, but where all replicas store enough information to answer every query. We apply this concept to eventual consistency and conflict-free replicated data types. We show that this model can address useful problems and present two data types that solve such problems. Our evaluation shows that non-uniform replication is more efficient than traditional replication, using less storage space and network bandwidth.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Non-uniform Replication, Partial Replication, Replicated Data Types, Eventual Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.24

1 Introduction

Many applications run on cloud infrastructures composed by multiple data centers, geographically distributed across the world. These applications usually store their data on geo-replicated data stores, with replicas of data being maintained in multiple data centers. Data management in geo-replicated settings is challenging, requiring designers to make a number of choices to better address the requirements of applications.

One well-known trade-off is between availability and data consistency. Some data stores provide strong consistency [5, 17], where the system gives the illusion that a single replica exists. This requires replicas to coordinate for executing operations, with impact on the latency and availability of these systems. Other data stores [7, 11] provide high-availability and low latency by allowing operations to execute locally in a single data center eschewing a linearizable consistency model. These systems receive and execute updates in a single replica before asynchronously propagating the updates to other replicas, thus providing very low latency.

* This work has been partially funded by CMU-Portugal research project GoLocal Ref. CMUP-ERI/TIC/0046/2014, EU LightKone (grant agreement n.732505) and by FCT/MCT project NOVA-LINCS Ref. UID/CEC/04516/2013. Part of the computing resources used in this research were provided by a Microsoft Azure Research Award.

With the increase of the number of data centers available to applications and the amount of information maintained by applications, another trade-off is between the simplicity of maintaining all data in all data centers and the cost of doing so. Besides sharding data among multiple machines in each data center, it is often interesting to keep only part of the data in each data center to reduce the costs associated with data storage and running protocols that involve a large number of replicas. In systems that adopt a partial replication model [22, 25, 6], as each replica only maintains part of the data, it can only locally process a subset of the database queries. Thus, when executing a query in a data center, it might be necessary to contact one or more remote data centers for computing the result of the query.

In this paper we explore an alternative partial replication model, the non-uniform replication model, where each replica maintains only part of the data but can process all queries. The key insight is that for some data objects, not all data is necessary for providing the result of read operations. For example, an object that keeps the top-K elements only needs to maintain those top-K elements in every replica. However, the remaining elements are necessary if a remove operation is available, as one of the elements not in the top needs to be promoted when a top element is removed.

A top-K object could be used for maintaining the leaderboard in an online game. In such system, while the information for each user only needs to be kept in the data center closest to the user (and in one or two more for fault tolerance), it is important to keep a replica of the leaderboard in every data center for low latency and availability. Currently, for supporting such a feature, several designs could be adopted. First, the system could maintain an object with the results of all players in all replicas. While simple, this approach turns out to be needlessly expensive in both storage space and network bandwidth when compared to our proposed model. Second, the system could move all data to a single data center and execute the computation in that data center or use a data processing system that can execute computations over geo-partitioned data [10]. The result would then have to be sent to all data centers. This approach is much more complex than our proposal, and while it might be interesting when complex machine learning computations are executed, it seems to be an overkill in a number of situations.

We apply the non-uniform replication model to eventual consistency and Conflict-free Replicated Data Types [23], formalizing the model for an operation-based replication approach. We present two useful data type designs that implement such model. Our evaluation shows that the non-uniform replication model leads to high gains in both storage space and network bandwidth used for synchronization when compared with state-of-the-art replication based alternatives.

In summary, this paper makes the following contributions:

- The proposal of the non-uniform replication model, where each replica only keeps part of the data but enough data to reply to every query;
- The definition of non-uniform eventual consistency (NuEC), the identification of sufficient conditions for providing NuEC and a protocol that enforces such conditions relying on operation-based synchronization;
- Two useful replicated data type designs that adopt the non-uniform replication model (and can be generalized to use different filter functions);
- An evaluation of the proposed model, showing its gains in term of storage space and network bandwidth.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the non-uniform replication model. Section 4 applies the model to an eventual consistent system. Section 5 introduces two useful data type designs that follow the model. Section 6 compares our proposed data types against state-of-the-art CRDTs.

2 Related Work

Replication. A large number of replication protocols have been proposed in the last decades [8, 27, 15, 16, 2, 21, 17]. Regarding the contents of the replicas, these protocols can be divided in those providing full replication, where each replica maintains the full database state, and partial replication, where each replica maintains only a subset of the database state.

Full replication strategies allow operations to concurrently modify all replicas of a system and, assuming that replicas are mutually consistent, improves availability since clients may query any replica in the system and obtain an immediate response. While this improves the performance of read operations, update operations now negatively affect the performance of the system since they must modify every replica which severely affects middle-scale to large-scale systems in geo-distributed settings. This model also has the disadvantage of limiting the system's total capacity to the capacity of the node with fewest resources.

Partial replication [3, 22, 25, 6] addresses the shortcomings of full replication by having each replica store only part of the data (which continues being replicated in more than one node). This improves the scalability of the system but since each replica maintains only a part of the data, it can only locally process a subset of queries. This adds complexity to the query processing, with some queries requiring contacting multiple replicas to compute their result. In our work we address these limitations by proposing a model where each replica maintains only part of the data but can reply to any query.

Despite of adopting full or partial replication, replication protocols enforce strong consistency [17, 5, 18], weak consistency [27, 7, 15, 16, 2] or a mix of these consistency models [24, 14]. In this paper we show how to combine non-uniform replication with eventual consistency. An important aspect in systems that adopt eventual consistency is how the system handles concurrent operations. CRDTs have been proposed as a technique for addressing such challenge.

CRDTs. Conflict-free Replicated Data Types [23] are data types designed to be replicated at multiple replicas without requiring coordination for executing operations. CRDTs encode merge policies used to guarantee that all replicas converge to the same value after all updates are propagated to every replica. This allows an operation to execute immediately on any replica, with replicas synchronizing asynchronously. Thus, a system that uses CRDTs can provide low latency and high availability, despite faults and network latency. With these guarantees, CRDTs are a key building block for providing eventual consistency with well defined semantics, making it easier for programmers to reason about the system evolution.

When considering the synchronization process, two main types of CRDTs have been proposed: state-based CRDT, where replicas synchronize pairwise, by periodically exchanging the state of the replicas; and operation-based CRDTs, where all operations need to be propagated to all replicas.

Delta-based CRDTs [1] improve upon state-based CRDTs by reducing the dissemination cost of updates, sending only a delta of the modified state. This is achieved by using *delta-mutators*, which are functions that encode a delta of the state. Linde et. al [26] propose an improvement to delta-based CRDTs that further reduce the data that need to be propagated when a replica first synchronizes with some other replica. This is particularly interesting in peer-to-peer settings, where the synchronization partners of each replica change frequently. Although delta-based CRDTs reduce the network bandwidth used for synchronization, they continue to maintain a full replication strategy where the state of quiescent replicas is equivalent.

Computational CRDTs [19] are an extension of state-based CRDTs where the state of the object is the result of a computation (e.g. the average, the top-K elements) over the executed updates. As with the model we propose in this paper, replicas do not need to have equivalent states. The work we present in this paper extends the initial ideas proposed in computational CRDTs in several aspects, including the definition of the non-uniform replication model, its application to operation-based eventual consistency and the new data type designs.

3 Non-uniform replication

We consider an asynchronous distributed system composed by n nodes. Without loss of generality, we assume that the system replicates a single object. The object has an interface composed by a set of read-only operations, $\mathcal{Q}$, and a set of update operations, $\mathcal{U}$. Let $\mathcal{S}$ be the set of all possible object states, the state that results from executing operation o in state $s \in \mathcal{S}$ is denoted as $s \bullet o$. For a read-only operation, $q \in \mathcal{Q}$, $s \bullet q = s$. The result of operation $o \in \mathcal{Q} \cup \mathcal{U}$ in state $s \in \mathcal{S}$ is denoted as $o(s)$ (we assume that an update operation, besides modifying the state, can also return some result).

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state of the replica i. The state of the replicas is synchronized by a replication protocol that exchanges messages among the nodes of the system and updates the state of the replicas. For now, we do not consider any specific replication protocol or strategy, as our proposal can be applied to different replication strategies.

We say a system is in a quiescent state for a given set of executed operations if the replication protocol has propagated all messages necessary to synchronize all replicas, i.e., additional messages sent by the replication protocol will not modify the state of the replicas. In general, replication protocols try to achieve a convergence property, in which the state of any two replicas is equivalent in a quiescent state.

▶ **Definition 1** (Equivalent state)**.** Two states, s_i and s_j, are *equivalent*, $s_i \equiv s_j$, iff the results of the execution of any sequence of operations in both states are equal, i.e., $\forall o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}, o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$.

This property is enforced by most replication protocols, independently of whether they provide strong or weak consistency [13, 15, 27]. We note that this property does not require that the internal state of the replicas is the same, but only that the replicas always return the same results for any executed sequence of operations.

In this work, we propose to relax this property by requiring only that the execution of read-only operations return the same value. We name this property as *observable equivalence* and define it formally as follows.

▶ **Definition 2** (Observable equivalent state)**.** Two states, s_i and s_j, are *observable equivalent*, $s_i \overset{\circ}{\equiv} s_j$, iff the result of executing every read-only operation in both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$.

As read-only operations do not affect the state of a replica, the results of the execution of any sequence of read-only operations in two observable equivalent states will also be the same. We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

▶ **Definition 3** (Non-uniform replicated system)**.** We say that a replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, the state of any two replicas is observable equivalent, i.e., in the quiescent state $(s_1, \ldots, s_n)$, we have $s_i \overset{\circ}{\equiv} s_j, \forall s_i, s_j \in \{s_1, \ldots, s_n\}$.

3.1 Example

We now give an example that shows the benefit of non-uniform replication. Consider an object *top-1* with three operations: (i) $add(name, value)$, an update operation that adds the pair to the top; (ii) $rmv(name)$, an update operation that removes all previously added pairs for *name*; (iii) $get()$, a query that returns the pair with the largest value (when more than one pair has the same largest value, the one with the smallest lexicographic name is returned).

Consider that $add(a, 100)$ is executed in a replica and replicated to all replicas. Later $add(b, 110)$ is executed and replicated. At this moment, all replicas know both pairs.

If later $add(c, 105)$ executes in some replica, the replication protocol does not need to propagate the update to the other replicas in a non-uniform replicated system. In this case, all replicas are observable equivalent, as a query executed at any replica returns the same correct value. This can have an important impact not only in the size of object replicas, as each replica will store only part of the data, but also in the bandwidth used by the replication protocol, as not all updates need to be propagated to all replicas.

We note that the states that result from the previous execution are not equivalent because after executing $rmv(b)$, the get operation will return $(c, 105)$ in the replica that has received the $add(c, 105)$ we operation and $(b, 100)$ in the other replicas.

Our definition only forces the states to be observable equivalent after the replication protocol becomes quiescent. Different protocols can be devised giving different guarantees. For example, for providing linearizability, the protocol should guarantee that all replicas return $(c, 105)$ after the remove. This can be achieved, for example, by replicating the now relevant $(c, 105)$ update in the process of executing the remove.

In the remainder of this paper, we study how to apply the concept of non-uniform replication in the context of eventually consistent systems. The study of its application to systems that provide strong consistency is left for future work.

4 Non-uniform eventual consistency

We now apply the concept of non-uniform replication to replicated systems providing eventual consistency.

4.1 System model

We consider an asynchronous distributed system composed by n nodes, where nodes may exhibit fail-stop faults but not byzantine faults. We assume a communication system with a single communication primitive, $mcast(m)$, that can be used by a process to send a message to every other process in the system with reliable broadcast semantics. A message sent by a correct process is eventually received by all correct processes. A message sent by a faulty process is either received by all correct processes or none. Several communication systems provide such properties – e.g. systems that propagate messages reliably using anti-entropy protocols [8, 9].

An object is defined as a tuple $(\mathcal{S}, s^0, \mathcal{Q}, \mathcal{U}_p, \mathcal{U}_e)$, where $\mathcal{S}$ is the set of valid states of the object, $s^0 \in \mathcal{S}$ is the initial state of the object, $\mathcal{Q}$ is the set of read-only operations (or *queries*), $\mathcal{U}_p$ is the set of prepare-update operations and $\mathcal{U}_e$ is the set of effect-update operations.

A query executes only at the replica where the operation is invoked, its source, and it has no side-effects, i.e., the state of an object remains unchanged after executing the operation.

When an application wants to update the state of the object, it issues a prepare-update operation, $u_p \in \mathcal{U}_p$. A u_p operation executes only at the source, has no side-effects and generates an effect-update operation, $u_e \in \mathcal{U}_e$. At source, u_e executes immediately after u_p.

As only effect-update operations may change the state of the object, for reasoning about the evolution of replicas we can restrict our analysis to these operations. To be precise, the execution of a prepare-update operation generates an instance of an effect-update operation. For simplicity, we refer the instances of operations simply as operations. With O_i the set of operations generated at node i, the set of operations generated in an execution, or simply the set of operations in an execution, is $O = O_1 \cup \ldots \cup O_n$.

4.2 Non-uniform eventual consistency

For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations of O; and (ii) the state of any pair of replicas is equivalent.

A sufficient condition for achieving the first property is to propagate all generated operations using reliable broadcast (and execute any received operation). A sufficient condition for achieving the second property is to have only commutative operations. Thus, if all operations commute with each other, the execution of any serialization of O in the initial state of the object leads to an equivalent state.

From now on, unless stated otherwise, we assume that all operations commute. In this case, as all serializations of O are equivalent, we denote the execution of a serialization of O in state s simply as $s \bullet O$.

For any given execution, with O the operations of the execution, we say a replicated system provides *non-uniform eventual consistency* iff in a quiescent state the state of any replica is observable equivalent to the state obtained by executing some serialization of O. As a consequence, the state of any pair of replicas is also observable equivalent.

For a given set of operations in an execution O, we say that $O_{core} \subseteq O$ is a set of core operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet O_{core}$. We define the set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

▶ **Theorem 4** (Sufficient conditions for NuEC). *A replication system provides* non-uniform eventual consistency (NuEC) *if, for a given set of operations O, the following conditions hold: (i) every replica executes a set of core operations of O; and (ii) all operations commute.*

Proof. From the definition of core operations of O, and by the fact that all operations commute, it follows immediately that if a replica executes a set of core operations, then the final state of the replica is observable equivalent to the state obtained by executing a serialization of O. Additionally, any replica reaches an observable equivalent state. ◀

4.3 Protocol for non-uniform eventual consistency

We now build on the sufficient conditions for providing *non-uniform eventual consistency* to devise a correct replication protocol that tries to minimize the operations propagated to other replicas. The key idea is to avoid propagating operations that are part of a masked set. The challenge is to achieve this by using only local information, which includes only a subset of the executed operations.

Algorithm 1 presents the pseudo-code of an algorithm for achieving *non-uniform eventual consistency* – the algorithm does not address the durability of operations, which will be discussed later.

Algorithm 1 Replication algorithm for non-uniform eventual consistency.

```
1:  S : state: initial s^0                                          ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                       ▷ Local operations not propagated
4:
5:  execOp(op): void                                                ▷ New operation generated locally
6:    log_local = log_local ∪ {op}
7:    S = S • op
8:
9:  opsToPropagate(): set of operations ▷ Computes the local operations that need to be propagated
10:   ops = maskedForever(log_local, S, log_recv)
11:   log_local = log_local \ ops
12:   opsImpact = hasObservableImpact(log_local, S, log_recv)
13:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
14:   return opsImpact ∪ opsPotImpact
15:
16: sync(): void                                                    ▷ Propagates local operations to remote replicas
17:   ops = opsToPropagate()
18:   compactedOps = compact(ops)                                   ▷ Compacts the set of operations
19:   mcast(compactedOps)
20:   log_coreLocal = {}
21:   log_local = log_local \ ops
22:   log_recv = log_recv ∪ ops
23:
24: on receive(ops): void                                           ▷ Process remote operations
25:   log_recv = log_recv ∪ ops
26:   S = S • ops
```

The algorithm maintains the state of the object and two sets of operations: log_{local}, the set of effect-update operations generated in the local replica and not yet propagated to other replicas; log_{recv}, the set of effect-update operations propagated to all replicas (including operations generated locally and remotely).

When an effect-update operation is generated, the *execOp* function is called. This function adds the new operation to the log of local operations and updates the local object state.

The function *sync* is called to propagate local operations to remote replicas. It starts by computing which new operations need to be propagated, compacts the resulting set of operations for efficiency purposes, multicasts the compacted set of operations, and finally updates the local sets of operations. When a replica receives a set of operations (line 24), the set of operations propagated to all nodes and the local object state are updated accordingly.

Function *opsToPropagate* addresses the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of the operations that might be executed in the future. In the top example, an operation that adds a pair masks forever all known operations that added a pair for the same element with a lower value. These operations are removed from the set of local operations.

Second, the *core* operations (*opsImpact*, line 12), as computed locally. These operations need to be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considered in combination with other non-core operations that might have been executed in other replicas (*opsPotImpact*, line 13). As there is no way to know which non-core operations have been executed in other replicas, it is necessary to propagate these operations also. For example, consider a modified top object where the value associated with each element is the sum of the values of the pairs added to the object. In this case, an add operation that would not move an element to the top in a replica would be in this category because it could influence the top when combined with other concurrent adds for the same element.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state. These operations remain in log_{local}. In the original top example, an operation that adds a pair that will not be in the top, as computed locally, is in this category as it might become the top element after removing the elements with larger values.

For proving that the algorithm can be used to provide non-uniform eventual consistency, we need to prove the following property.

▶ **Theorem 5.** *Algorithm 1 guarantees that in a quiescent state, considering all operations O in an execution, all replicas have received all operations in a core set O_{core}.*

Proof. To prove this property, we need to prove that there exists no operation that has not been propagated by some replica and that is required for any O_{core} set. Operations in the first category have been identified as masked operations independently of any other operations that might have been or will be executed. Thus, by definition of masked operations, a O_{core} set will not (need to) include these operations. The fourth category includes operations that do not influence the observable state when considering all executed operations – if they might have impact, they would be in the third category. Thus, these operations do not need to be in a O_{core} set. All other operations are propagated to all replicas. Thus, in a quiescent state, every replica has received all operations that impact the observable state. ◀

4.4 Fault-tolerance

Non-uniform replication aims at reducing the cost of communication and the size of replicas, by avoiding propagating operations that do not influence the observable state of the object. This raises the question of the durability of operations that are not immediately propagated to all replicas.

One way to solve this problem is to have the source replica propagating every local operation to f more replicas to tolerate f faults. This ensures that an operation survives even in the case of f faults. We note that it would be necessary to adapt the proposed algorithm, so that in the case a replica receives an operation for durability reasons, it would propagate the operation to other replicas if the source replica fails. This can be achieved by considering it as any local operation (and introducing a mechanism to filter duplicate reception of operations).

4.5 Causal consistency

Causal consistency is a popular consistency model for replicated systems [15, 2, 16], in which a replica only executes an operation after executing all operations that causally precede it [12]. In the non-uniform replication model, it is impossible to strictly adhere to this definition because some operations are not propagated (immediately), which would prevent all later operations from executing.

An alternative would be to restrict the dependencies to the execution of core operations. The problem with this is that the status of an operation may change by the execution of another operation. When a non-core operation becomes core, a number of dependencies that should have been enforced might have been missed in some replicas.

We argue that the main interest of causal consistency, when compared with eventual consistency, lies in the semantics provided by the object. Thus, in the designs that we present in the next section, we aim to guarantee that in a quiescent state, the state of the replicated objects provide equivalent semantics to that of a system that enforces causal consistency.

5 Non-uniform operation-based CRDTs

CRDTs [23] are data-types that can be replicated, modified concurrently without coordination and guarantee the eventual consistency of replicas given that all updates propagate to all replicas. We now present the design of two useful operation-based CRDTs [23] that adopt the non-uniform replication model. Unlike most operation-based CRDT designs, we do not assume that the system propagates operations in a causal order. These designs were inspired by the state-based computational CRDTs proposed by Navalho *et al.* [19], which also allow replicas to diverge in their quiescent state.

5.1 Top-K with removals NuCRDT

In this section we present the design of a non-uniform top-K CRDT, as the one introduced in section 3.1. The data type allows access to the top-K elements added and can be used, for example, for maintaining the leaderboard in online games. The proposed design could be adapted to define any CRDT that filters elements based on a deterministic function by replacing the *topK* function used in the algorithm by another filter function.

For defining the semantics of our data type, we start by defining the happens-before relation among operations. To this end, we start by considering the happens-before relation established among the events in the execution of the replicated system [12]. The events that are considered relevant are: the generation of an operation at the source replica, and the dispatch and reception of a message with a new operation or information that no new message exists. We say that operation op_i happens before operation op_j iff the generation of op_i happened before the generation of op_j in the partial order of events.

The semantics of the operations defined in the top-K CRDT is the following. The *add(el,val)* operation adds a new pair to the object. The *rmv(el)* operation removes any pair of *el* that was added by an operation that happened-before the *rmv* (note that this includes non-core add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [23], where a remove has no impact on concurrent adds. The *get()* operation returns the top-K pairs in the object, as defined by the function *topK* used in the algorithm.

Algorithm 2 presents a design that implements this semantics. The prepare-update *add* operation generates an effect-update *add* that has an additional parameter consisting in a timestamp $(replicaid, val)$, with *val* a monotonically increasing integer. The prepare-update *rmv* operation generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes add operations that happened before the remove operation. To this end, the object maintains a vector clock that is updated when a new add is generated or executed locally. Additionally, this vector clock should be updated whenever a replica receives a message from a remote replica (to summarize also the adds known in the sender that have not been propagated to this replica).

Besides this vector clock, *vc*, each object replica maintains: (i) a set, *elems*, with the elements added by all *add* operations known locally (and that have not been removed yet); and (ii) a map, *removes*, that maps each element *id* to a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of *elems* if the add has not happened before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a *rmv* consists

Algorithm 2 Top-K NuCRDT with removals.

```
1:  elems : set of ⟨id, score, ts⟩ : initial {}
2:  removes : map id ↦ vectorClock: initial []
3:  vc : vectorClock: initial []
4:
5:  GET() : set
6:    return {⟨id, score⟩ : ⟨id, score, ts⟩ ∈ topK(elems)}
7:
8:  prepare ADD(id, score)
9:    generate add(id, score, ⟨getReplicaId(), ++ vc[getReplicaId()]⟩)
10:
11: effect ADD(id, score, ts)
12:   if removes[id][ts.siteId] < ts.val then
13:     elems = elems ∪ {⟨id, score, ts⟩}
14:     vc[ts.siteId] = max(vc[ts.siteId], ts.val)
15:
16: prepare RMV(id)
17:   generate rmv(id,vc)
18:
19: effect RMV(id, vc_rmv)
20:   removes[id] = pointwiseMax(removes[id], vc_rmv)
21:   elems = elems \ {⟨id_0, score, ts⟩ ∈ elem : id = id_0 ∧ ts.val ≤ vc_rmv[ts.siteId]}
22:
23: maskedForever(log_local, S, log_recv): set of operations
24:   adds = {add(id_1, score_1, ts_1) ∈ log_local :
25:     (∃add(id_2, score_2, ts_2) ∈ log_local : id_1 = id_2 ∧ score_1 < score_2 ∧ ts_1.val < ts_2.val)∨
26:       (∃rmv(id_3, vc_rmv) ∈ (log_recv ∪ log_local) : id_1 = id_3 ∧ ts_1.val ≤ vc_rmv[ts_1.siteId]}
27:   rmvs = {rmv(id_1, vc_1) ∈ log_local : ∃rmv(id_2, vc_2) ∈ (log_local ∪ log_recv) : id_1 = id_2 ∧ vc_1 < vc_2}
28:   return adds ∪ rmvs
29:
30: mayHaveObservableImpact(log_local, S, log_recv): set of operations
31:   return {}                                  ▷ This case never happens for this data type
32:
33: hasObservableImpact(log_local, S, log_recv): set of operations
34:   adds = {add(id_1, score_1, ts_1) ∈ log_local : ⟨id_1, score_1, ts_1⟩ ∈ topK(S.elems)}
35:   rmvs = {rmv(id_1, vc_1) ∈ log_local : (∃add(id_2, score_2, ts_2) ∈ (log_local ∪ log_recv) :
36:       ⟨id_2, score_2, ts_2⟩ ∈ topK(S.elems ∪ {⟨id_2, score_2, ts_2⟩}) ∧ id_1 = id_2 ∧ ts_2.val ≤ vc_1[ts_2.siteId])}
37:   return adds ∪ rmvs
38:
39: compact(ops): set of operations
40:   return ops                                 ▷ This data type does not require compaction
```

in updating *removes* and deleting from *elems* the information for adds of the element that happened before the remove. To verify if an add has happened before a remove, we check if the timestamp associated with the add is reflected in the remove vector clock of the element (lines 12 and 21). This ensures the intended semantics for the CRDT, assuming that the functions used by the protocol are correct.

We now analyze the code of these functions.

Function maskedForever computes: the local adds that become masked by other local adds (those for the same element with a lower value) and removes (those for the same element that happened before the remove); the local removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing the one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element but with a larger value as it is not possible to remove only the effects of the later add without removing the effect of the older one. A local add also becomes permanently masked by a local or remote remove that happened after the add.

Function mayHaveObservableImpact returns the empty set, as for having impact on any observable state, an operation also has to have impact on the local observable state by itself.

Function hasObservableImpact computes the local operations that are relevant for computing the top-K. An add is relevant if the added value is in the top; a remove is relevant if it removes an add that would be otherwise in the top.

5.2 Top Sum NuCRDT

We now present the design of a non-uniform CRDT, Top Sum, that maintains the top-K elements added to the object, where the value of each element is the sum of the values added for the element. This data type can be used for maintaining a leaderboard in an online game where every time a player completes some challenge it is awarded some number of points, with the current score of the player being the sum of all points awarded. It could also be used for maintaining a top of the best selling products in an (online) store (or the top customers, etc).

The semantics of the operations defined in the Top Sum object is the following. The *add(id, n)* update operation increments the value associated with *id* by *n*. The *get()* read-only operation returns the top-K mappings, $id \rightarrow value$, as defined by the *topK* function (similar to the Top-K NuCRDT).

This design is challenging, as it is hard to know which operations may have impact in the observable state. For example, consider a scenario with two replicas, where the value of the last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations.

To try to minimize the number of operations propagated we use the following heuristic inspired by the demarcation protocol and escrow transactions [4, 20]. For each id that does not belong to the top, we compute the difference between the smallest value in the top and the value of the id computed by operations known in every replica – this is how much must be added to the id to make it to the top: let d be this value. If the sum of local adds for the id does not exceed $\frac{d}{num.replicas}$ in any replica, the value of id when considering adds executed in all replicas is smaller than the smallest element in the top. Thus, it is not necessary to propagate add operations in this case, as they will not affect the top.

Algorithm 3 presents a design that implements this approach. The state of the object is a single variable, *state*, that maps identifiers to their current values. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. The execution of an effect-update *add(id, n)* simply increments the value of *id* by *n*.

Function maskedForever returns the empty set, as operations in this design can never be forever masked.

Function mayHaveObservableImpact computes the set of *add* operations that can potentially have an impact on the observable state, using the approach previously explained.

Function hasObservableImpact computes the set of *add* operations that have their corresponding *id* present in the top-K. This guarantees that the values of the elements in the top are kept up-to-date, reflecting all executed operations.

Function compact takes a set of *add* operations and compacts the *add* operations that affect the same identifier into a single operation. This reduces the size of the messages sent through the network and is similar to the optimization obtained in delta-based CRDTs [1].

Algorithm 3 Top Sum NuCRDT.

```
1: state : map id ↦ sum: initial []
2:
3: GET() : map
4:   return topK(state)
5:
6: prepare ADD(id, n)
7:   generate add(id, n)
8:
9: effect ADD(id, n)
10:  state[id] = state[id] + n
11:
12: MASKEDFOREVER(log_local, S, log_recv): set of operations
13:  return {}                                  ▷ This case never happens for this data type
14:
15: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
16:  top = topK(S.state)
17:  adds = {add(id, _) ∈ log_local : s = sum_val({add(i, n) ∈ log_local : i = id})
18:    ∧ s ≥ ((min(sum(top)) − (S.state[id] − s)) / getNumReplicas())}
19:  return adds
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
22:  top = topK(S.state)
23:  adds = {add(id, _) ∈ log_local : id ∈ ids(top)}
24:  return adds
25:
26: COMPACT(ops): set of operations
27:  adds = {add(id, n) : id ∈ {i : add(i, _) ∈ ops}∧ n = sum({k : add(id_1, k) ∈ ops : id_1 = id})}
28:  return adds
```

5.3 Discussion

The goal of non-uniform replication is to allow replicas to store less data and use less bandwidth for replica synchronization. Although it is clear that non-uniform replication cannot be useful for all data, we believe that the number of use cases is large enough for making non-uniform replication interesting in practice. We now discuss two classes of data types that can benefit from the adoption of non-uniform replication.

The first class is that of data types for which the result of queries include only a subset of the data in the object. In this case two different situations may occur: (i) it is possible to compute locally, without additional information, if some operation is relevant (and needs to be propagated to all replicas); (ii) it is necessary to have additional information to be able to decide if some operation is relevant. The Top-K CRDT presented in section 5.1 is an example of the former. Another example includes a data type that returns a subset of the elements added based on a (modifiable) user-defined filter – e.g. in a set of books, the filter could select the books of a given genre, language, etc. The Top-Sum CRDT presented in section 5.2 is an example of the latter. Another example includes a data type that returns the 50^{th} percentile (or others) for the elements added – in this case, it is only necessary to replicate the elements in a range close to the 50^{th} percentile and replicate statistics of the elements smaller and larger than the range of replicated elements.

In all these examples, the effects of an operation that in a given moment do not influence the result of the available queries may become relevant after other operations are executed – in the Top-K with removes due to a remove of an element in the top; in the filtered set due to a change in the filter; in the Top-Sum due to a new add that makes an element relevant; and in the percentile due to the insertion of elements that make the 50^{th} percentile change. We note that if the relevance of an operation cannot change over time, the non-uniform CRDT would be similar to an optimized CRDT that discard operations that are not relevant before propagating them to other replicas.

A second class is that of data types with queries that return the result of an aggregation over the data added to the object. An example of this second class is the Histogram CRDT presented in the appendix. This data type only needs to keep a count for each element. A possible use of this data type would be for maintaining the summary of classifications given by users in an online shop. Similar approaches could be implemented for data types that return the result of other aggregation functions that can be incrementally computed [19].

A data type that supports, besides adding some information, an operation for removing that information would be more complex to implement. For example, in an Histogram CRDT that supports removing a previously added element, it would be necessary that concurrently removing the same element would not result in an incorrect aggregation result. Implementing such CRDT would require detecting and fixing these cases.

6 Evaluation

In this section we evaluate our data types that follow the non-uniform replication model. To this end, we compare our designs against state-of-the-art CRDT alternatives: delta-based CRDTs [1] that maintain full object replicas efficiently by propagating updates as deltas of the state; and computational CRDTs [19] that maintain non-uniform replicas using a state-based approach.

Our evaluation is performed by simulation, using a discrete event simulator. To show the benefit in terms of bandwidth and storage, we measure the total size of messages sent between replicas for synchronization (total payload) and the average size of replicas.

We simulate a system with 5 replicas for each object. Both our designs and the computational CRDTs support up to 2 replica faults by propagating all operations to, at least, 2 other replicas besides the source replica. We note that this limits the improvement that our approach could achieve, as it is only possible to avoid sending an operation to two of the five replicas. By either increasing the number of replicas or reducing the fault tolerance level, we could expect that our approach would perform comparatively better than the delta-based CRDTs.

6.1 Top-K with removals

We begin by comparing our Top-K design (*NuCRDT*) with a delta-based CRDT set [1] (*Delta CRDT*) and the top-K state-based computational CRDT design [19] (*CCRDT*).

The top-K was configured with K equal to 100. In each run, 500000 update operations were generated for 10000 Ids and with scores up to 250000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Given the expected usage of top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Figures 1 and 2 show the results for workloads with 5% and 0.05% of removes respectively (the other operations are adds).

In both workloads our design achieves a significantly lower bandwidth cost when compared to the alternatives. The reason for this is that our design only propagates operations that will be part of the top-K. In the delta-based CRDT, each replica propagates all new updates and not only those that are part of the top. In the computational CRDT design, every time the top is modified, the new top is propagated. Additionally, the proposed design of computational CRDTs always propagates removes.

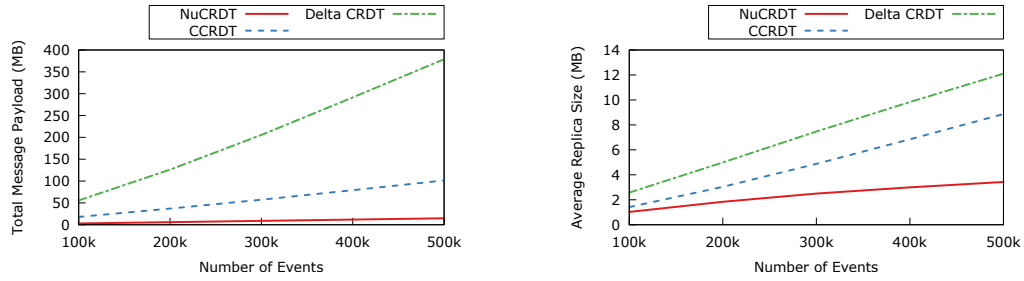


Figure 1 Top-K with removals: payload size and replica size, workload of 95/5

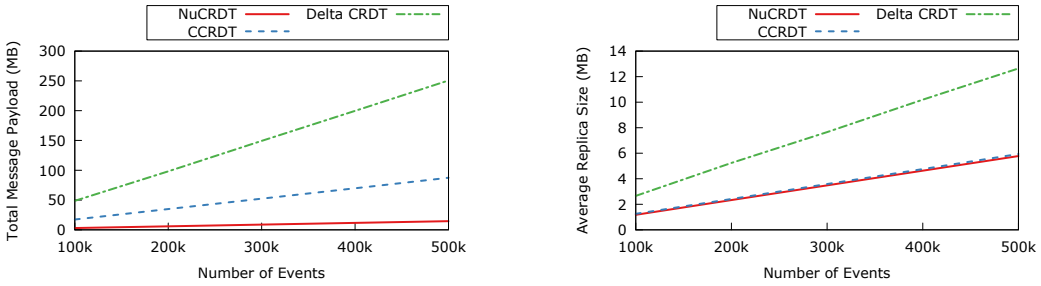


Figure 2 Top-K with removals: payload size and replica size, workload of 99.95/0.05

The results for the replica size show that our design is also more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information from remote operations received for guaranteeing fault-tolerance and those that have influenced the top-K at some moment in the execution. The computational CRDT design additionally keeps information about all removes. The delta-based CRDT keeps information about all elements that have not been removed or overwritten by a larger value. We note that as the percentage of removes approaches zero, the replica sizes of our design and that of computational CRDT starts to converge to the same value. The reason for this is that the information maintained in both designs is similar and our more efficient handling of removes starts becoming irrelevant. The opposite is also true: as the number of removes increases, our design becomes even more space efficient when compared to the computational CRDT.

6.2 Top Sum

To evaluate our Top Sum design (*NuCRDT*), we compare it against a delta-based CRDT map (*Delta CRDT*) and a state-based computational CRDT implementing the same semantics (*CCRDT*).

The top is configured to display a maximum of 100 entries. In each run, 500000 update operations were generated for 10000 Ids and with challenges awarding scores up to 1000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Figure 3 shows the results of our evaluation. Our design achieves a significantly lower bandwidth cost when compared with the computational CRDT, because in the computational CRDT design, every time the top is modified, the new top is propagated. When compared with the delta-based CRDTs, the bandwidth of NuCRDT is approximately 55% of the bandwidth used by delta-based CRDTs. As delta-based CRDTs also include a mechanism for compacting propagated updates, the improvement comes from the mechanisms for avoiding propagating operations that will not affect the top elements, resulting in less messages being sent.

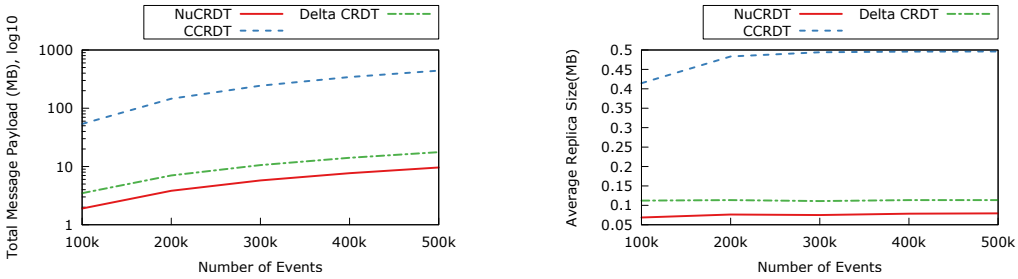


Figure 3 Top Sum: payload size and replica size

The results for the replica size show that our design also manages to be more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information of remote operations received for guaranteeing fault-tolerance and those that have influenced the top elements at some moment in the execution.

7 Conclusions

In this paper we proposed the non-uniform replication model, an alternative model for replication that combines the advantages of both full replication, by allowing any replica to reply to a query, and partial replication, by requiring that each replica keeps only part of the data. We have shown how to apply this model to eventual consistency, and proposed a generic operation-based synchronization protocol for providing non-uniform replication. We further presented the designs of two useful replicated data types, the Top-K and Top Sum, that adopt this model (in appendix, we present two additional designs: Top-K without removals and Histogram). Our evaluation shows that the application of this new replication model helps to reduce the message dissemination costs and the size of replicas.

In the future we plan to study which other data types can be designed that adopt this model and to study how to integrate these data types in cloud-based databases. We also want to study how the model can be applied to strongly consistent systems.

References

1 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018. doi:10.1016/j.jpdc.2017.08.003.

2 Sérgio Almeida, João Leitão, and Luís E. T. Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 85–98. ACM, 2013. doi:10.1145/2465351.2465361.

3 Gustavo Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*, 1997.

4 Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, jul 1994. doi:10.1007/BF01232643.

5 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi

Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

6 Tyler Crain and Marc Shapiro. Designing a causally consistent protocol for geo-distributed partial replication. In Carlos Baquero and Marco Serafini, editors, *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2015, Bordeaux, France, April 21, 2015*, pages 6:1–6:4. ACM, 2015. doi:10.1145/2745947.2745953.

7 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007. doi:10.1145/1294261.1294281.

8 Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In Fred B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 1–12. ACM, 1987. doi:10.1145/41840.41841.

9 Patrick Th. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, 2004. doi:10.1109/MC.2004.1297243.

10 Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics. *Proc. VLDB Endow.*, 9(2):72–83, 2015. doi:10.14778/2850578.2850582.

11 Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010. doi:10.1145/1773912.1773922.

12 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

13 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

14 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, 2012.

15 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416. ACM, 2011. doi:10.1145/2043556.2043593.

16 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

17 Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9), jul 2013. doi:10.14778/2536360.2536366.

18 Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Blotter: Low latency transactions for geo-replicated storage. In Rick Barrett, Rick Cummings, Eugene Agichtein, and Evgeniy Gabrilovich, editors, *Proceedings*

of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017, pages 263–272. ACM, 2017. doi:10.1145/3038912.3052603.

19 David Navalho, Sérgio Duarte, and Nuno M. Preguiça. A study of crdts that do computations. In Carlos Baquero and Marco Serafini, editors, *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2015, Bordeaux, France, April 21, 2015*, pages 1:1–1:4. ACM, 2015. doi:10.1145/2745947.2745948.

20 Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, 1986. doi:10.1145/7239.7265.

21 Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005. doi:10.1145/1057977.1057980.

22 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 214–224. IEEE Computer Society, 2010. doi:10.1109/SRDS.2010.32.

23 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, 2011.

24 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM, 2011. doi:10.1145/2043556.2043592.

25 Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 309–324. ACM, 2013. doi:10.1145/2517349.2522731.

26 Albert van der Linde, João Leitão, and Nuno M. Preguiça. Δ-crdts: making Δ-crdts delta-based. In Peter Alvaro and Alysson Bessani, editors, *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 12:1–12:4. ACM, 2016. doi:10.1145/2911151.2911163.

27 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009. doi:10.1145/1435417.1435432.

A Appendix

In this appendix we present two additional NuCRDT designs. These designs exemplify the use of different techniques for the creation of NuCRDTs.

A.1 Top-K without removals

A simpler example of a data type that fits our proposed replication model is a plain top-K, without support for the remove operation. This data type allows access to the top-K elements added to the object and can be used, for example, for maintaining a leaderboard in an online game. The top-K defines only one update operation, *add(id,score)*, which adds element *id* with score *score*. The *get()* operation simply returns the K elements with largest scores. Since the data type does not support removals, and elements added to the top-K which do not fit will simply be discarded this means the only case where operations have an impact in the observable state are if they are core operations – i.e. they are part of the top-K. This greatly simplifies the non-uniform replication model for the data type.

Algorithm 4 Top-K NuCRDT.

```
1: elems : {⟨id, score⟩} : initial {}
2:
3: GET() : set
4:   return elems
5:
6: prepare ADD(id, score)
7:   generate add(id, score)
8:
9: effect ADD(id, score)
10:  elems = topK(elems ∪ {⟨id, score⟩})
11:
12: MASKEDFOREVER(log_local, S, log_recv) : set of operations
13:  adds = {add(id_1, score_1) ∈ log_local : (∃add(id_2, score_2) ∈ log_recv : id_1 = id_2 ∧ score_2 > score_1)
14:  return adds
15:
16: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
17:  return {}                                   ▷ Not required for this data type
18:
19: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
20:  return {add(id, score) ∈ log_local : ⟨id, score⟩ ∈ S.elems}
21:
22: COMPACT(ops): set of operations
23:  return ops                                  ▷ This data type does not use compaction
```

Algorithm 4 presents the design of the top-K NuCRDT. The prepare-update *add(id,score)* generates an effect-update *add(id,score)*.

Each object replica maintains only a set of K tuples, *elems*, with each tuple being composed of an *id* and a *score*. The execution of *add(id,score)* inserts the element into the set, *elems*, and computes the top-K of *elems* using the function *topK*. The order used for the *topK* computation is as follows: $\langle id_1, score_1\rangle > \langle id_2, score_2\rangle$ iff $score_1 > score_2 \vee (score_1 = score_2 \wedge id_1 > id_2)$. We note that the *topK* function returns only one tuple for each element *id*.

Function MASKEDFOREVER computes the adds that become masked by other add operations for the same *id* that are larger according to the defined ordering. Due to the way the top is computed, the lower values for some given *id* will never be part of the top. Function MAYHAVEOBSERVABLEIMPACT always returns the empty set since operations in this data type are always core or forever masked. Function HASOBSERVABLEIMPACT returns the set of unpropagated add operations which add elements that are part of the top – essentially, the add operations that are core at the time of propagation. Function COMPACT simply returns the given *ops* since the design does not require compaction.

A.2 Histogram

We now introduce the Histogram NuCRDT that maintains a histogram of values added to the object. To this end, the data type maintains a mapping of bins to integers and can be used to maintain a voting system on a website. The semantics of the operations defined in the histogram is the following: *add(n)* increments the bin *n* by 1; *merge(histogram$_{delta}$)* adds the information of a histogram into the local histogram; *get()* returns the current histogram.

This data type is implemented in the design presented in Algorithm 5. The prepare-update *add(n)* generates an effect-update *merge($[n \mapsto 1]$)*. The prepare-update operation *merge(histogram)* generates an effect-update *merge(histogram)*.

Each object replica maintains only a map, *histogram*, which maps *bins* to integers. The execution of a *merge(histogram$_{delta}$)* consists of doing a pointwise sum of the local histogram with *histogram$_{delta}$*.

Algorithm 5 Histogram NuCRDT.

```
1:  histogram : map bin ↦ n : initial []
2:
3:  GET() : map
4:    return histogram
5:
6:  prepare ADD(bin)
7:    generate merge([bin ↦ 1])
8:
9:  prepare MERGE(histogram)
10:   generate merge(histogram)
11:
12: effect MERGE(histogram_delta)
13:   histogram = pointwiseSum(histogram, histogram_delta)
14:
15: MASKEDFOREVER(log_local, S, log_recv) : set of operations
16:   return {}                                  ▷ Not required for this data type
17:
18: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
19:   return {}                                  ▷ Not required for this data type
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
22:   return log_local
23:
24: COMPACT(ops): set of operations
25:   deltas = {hist : merge(hist_delta) ∈ ops}
26:   return {merge(pointwiseSum(deltas))}
```

Functions MASKEDFOREVER and MAYHAVEOBSERVABLEIMPACT always return the empty set since operations in this data type are always core. Function HASOBSERVABLEIMPACT simply returns log_{local}, as all operations are core in this data type. Function COMPACT takes a set of instances of *merge* operations and joins the histograms together returning a set containing only one *merge* operation.

Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus*

Ittai Abraham[1], Dahlia Malkhi[2], Kartik Nayak[3], Ling Ren[4], and Alexander Spiegelman[5]

1 **VMware Research, Palo Alto, USA**
`iabraham@vmware.com`
2 **VMware Research, Palo Alto, USA**
`dmalkhi@vmware.com`
3 **University of Maryland, College Park, USA**
`kartik@cs.umd.edu`
4 **Massachusetts Institute of Technology, Cambridge, USA**
`renling@mit.edu`
5 **Technion, Haifa, Israel**
`sasha.speigelman@gmail.edu`

Abstract

The decentralized cryptocurrency Bitcoin has experienced great success but also encountered many challenges. One of the challenges has been the long confirmation time. Another challenge is the lack of incentives at certain steps of the protocol, raising concerns for transaction withholding, selfish mining, etc. To address these challenges, we propose Solida, a decentralized blockchain protocol based on reconfigurable Byzantine consensus augmented by proof-of-work. Solida improves on Bitcoin in confirmation time, and provides safety and liveness assuming the adversary control less than (roughly) one-third of the total mining power.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Cryptocurrency, Blockchain, Byzantine fault tolerance, Reconfiguration

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.25

1 Introduction

Bitcoin is the most successful decentralized cryptocurrency to date. Conceptually, what a decentralized cryptocurrency needs is consensus in a *permissionless* setting: Participants should agree on the history of transactions, and anyone on the network can join or leave at any time. Bitcoin achieves permissionless consensus using what's now known as Nakamoto consensus. In Nakamoto consensus, participants accept the longest proof-of-work (PoW) chain as the history of transactions, and also contribute to the longest chain by trying to extend it. Thus, cryptocurrencies are also known as public ledgers or blockchains in the literature. While enjoying great success, Bitcoin does have several drawbacks. The most severe one is perhaps its limited throughput and long confirmation time of transactions. For instance, presently a block can be added every ten minutes on average and it is suggested that one waits for the transaction to be six blocks deep. This implies a confirmation time of about an hour. Furthermore, each block can contain about 1500 transactions [7] with the current 1MB block size limit. This yields a throughput of ∼2.5 transactions per second.

* This work is funded in part by NSF award #1518765, and a Google Ph.D. Fellowship award.

A number of attempts were made to improve the throughput and confirmation time of Bitcoin. These fall into two broad categories. One category [32, 19, 9] tries to improve Nakamoto consensus. The other category [8, 15, 21, 26] hopes to replace Nakamoto consensus with classical Byzantine fault tolerant (BFT) consensus protocols. These proposals envision a rolling committee that approves transactions efficiently using a Byzantine Fault Tolerant (BFT) protocol such as PBFT [5].

This second category presents a new approach to blockchain designs and has potential for significant improvements in performance and scalability over Nakamoto consensus. This is especially true for transaction confirmation time, since decisions in Byzantine consensus are final once committed. Pass and Shi [26] formalized the above intuition with *responsiveness* notion for permissionless consensus protocols. A protocol is *responsive* if it commits transactions (possibly probabilistically) at the speed of the *actual* network delay, without being bottlenecked by hard-coded system parameters (e.g., 10 minutes for Bitcoin). In the same paper, Pass and Shi proposed the hybrid consensus protocol, which uses a (slow) Nakamoto PoW chain to determine the identities of committee members, and let the committee to approve transactions responsively.

In this work, we present Solida[1], a decentralized blockchain protocol based on reconfigurable Byzantine consensus. We were indeed inspired by the work of Pass and Shi [26]. Yet, Solida is conceptually very different from Bitcoin or hybrid consensus as we do not rely on Nakamoto consensus for any part of our protocol. Committee election and transaction processing are both done by a Byzantine consensus protocol in Solida. PoW still plays a central role in Solida, but we remark that use of PoW does not equate Nakamoto consensus. The heart of Nakamoto consensus is the idea that "the longest PoW chain wins". This creates possibility of temporary "chain forks". A decision in Nakamoto consensus becomes committed only if it is "buried" sufficiently deep in the PoW chain, which leads to Bitcoin's long confirmation time.

Looking from a different angle, we can think of PoW as a leader election oracle that is Sybil-proof in the permissionless setting. But this oracle is imperfect as leader contention can still occur when multiple miners find PoWs around the same time. Nakamoto consensus can be thought of as a probabilistic method to resolve leader contention in which miners "vote on" contending leaders using their mining power. A leader (or its block) "wins" the contention gradually and probabilistically, until an overwhelming majority of miners "adopt" it by mining on top of it. In Solida, we also use PoW as an imperfect Sybil-proof leader election oracle. But instead of using Nakamoto consensus, we use a traditional Byzantine consensus protocol to resolve leader contention with certainty, rather than probabilistically.

Once we have a committee, electing new members using Byzantine consensus (as opposed to Nakamoto) just seems to be the natural design once we have a closer look at the details of the protocol. A central challenge of this framework is to reconcile the *permissioned* nature of Byzantine consensus protocols and the *permissionless* requirement of decentralized blockchains. Byzantine consensus protocols like PBFT assume a static group of committee members, but to be permissionless and decentralized, committee members must change over time – a step commonly referred to as *reconfiguration*. Note that the Nakamoto chain in hybrid consensus only provides agreement on the identities of the new members and when reconfiguration *should* happen. It does not dictate when and how reconfiguration *actually* happens. The standard reconfiguration technique requires a consensus decision from the old committee on the *closing state* [17, 13, 30, 3, 24, 1]. It is then just natural to include in that consensus decision the *new configuration* (i.e., the identity of the new member), at no extra cost, thereby eliminating the need for Nakamoto chains altogether.

[1] Solidus was a gold coin used in the Byzantine Empire. Solida is our way of (mis-)spelling Solidus.

Not relying on Nakamoto consensus gives Solida a few advantages over hybrid consensus. First, the identities of committee members are exposed for a shorter duration in Solida compared to hybrid consensus, because their identities no longer need to be buried in a Nakamoto chain. (Admittedly, members' identities are still exposed *during* their service on the committee for both Solida and hybrid consensus.) Second, defending selfish mining [10, 23] is much easier with the help of a committee (cf. Section 4.4). In comparison, hybrid consensus requires quite a complex variant of Nakamoto consensus in FruitChain [27] to defend against selfish mining. Third, analysis of our protocol is much simpler. In contrast, although the Bitcoin protocol is simple and elegant, its formal modeling and analysis turn out to be highly complex [12, 25]. FruitChain [27] and the interaction between Nakamoto and Byzantine consensus [26] further complicate the analysis.

Contribution. The high-level idea of designing blockchains protocols using reconfigurable Byzantine consensus is by no means new. PeerCensus [8] and ByzCoin [15] are two other works in this framework and they predate hybrid consensus and Solida. Comparing to those two works is tricky since their protocols were described only at a high level. But no matter how one interprets their protocols, our paper makes the following new contributions:

- **Detailed protocol.** We present full details of our protocol, and in particular, the reconfiguration step. Reconfiguration is perhaps the step that deserves the most detailed treatment, since the protocol between two reconfiguration events is just conventional Byzantine consensus.
- **Rigorous proof.** We rigorously prove that Solida achieves safety and liveness if the adversary's ratio of mining power does not exceed (roughly) 1/3.
- **Implementation and evaluation.** We implement Solida and measure its performance. Our implementation is fully Byzantine fault tolerant. With a 0.1s network latency and 75 Mbps network bandwidth, Solida with committees of 1000 members can commit a consensus decision in 23.6 seconds.

1.1 Overview of the Solida Protocol

At a high level, Solida runs a Byzantine consensus protocol among a set of participants, called a committee, that dynamically change over time. The acting committee commits transactions into a linearizable log using a modified version of PBFT [5]. The log is comparable to the Bitcoin blockchain, also called a public ledger. We say each consensus decision fills a *slot* in the ledger. A consensus decision can be either (1) a batch of transactions, (2) a reconfiguration event. The first type of decision is analogous to a block in Bitcoin. The second type records the membership change in the committee.

We denote the i-th member in chronological order as M_i, and the committee size is $n = 3f + 1$. Then, the i-th committee $C_i = (M_i, M_{i+1}, M_{i+2}, \cdots, M_{i+n-1})$. The first committee C_1 is known as the Genesis committee, and its n members are hard coded. After that, each new member is elected onto the committee one at a time, by the acting committee using *reconfiguration consensus decisions*.

At any time, one committee member serves as the leader. It proposes a batch of transactions into the next empty slot s in the ledger. Other members validate proposed transactions (no double spend, etc.) and commit them in two phases. A Byzantine leader cannot violate safety but may prevent progress. If members detect lack of progress, they elect the next leader in the round robin order using standard techniques from PBFT view change. The next leader needs to stop previous leaders, learn the status from $2f + 1$ members and possibly redo consensus for previous slots, before continuing on to future empty slots.

The more interesting part of protocol is reconfiguration. To be elected onto the committee, a miner needs to present a PoW, i.e., a solution to a moderately hard computational puzzle. Upon seeing this PoW, the current committee tries to reach a special *reconfiguration consensus decision* that commits (1) the identity of the new member, i.e., a public key associated with the PoW, and (2) the closing state before the reconfiguration. Once this reconfiguration consensus decision is committed, the system transitions into the new configuration C_{i+1}. Starting from the next slot, PoW finder becomes the newest committee member M_{i+n}, and the oldest committee member M_i loses its committee membership.

Here, a crucial design choice is: *who should be the leader that drives reconfiguration?* The first idea that comes to mind is to keep relying on the round robin leaders. But this approach presents a challenge on the analysis. The current leader may be Byzantine and refuse to reconfigure when it should. By doing so, it tries to buy time for other adversarial miners to find competing PoWs, so that it can nominate a Byzantine new member instead. Although we can conceive mechanisms to replace this leader, subsequent leaders may also be Byzantine. The probability of taking in an honest new member now depends on the pattern of consecutive Byzantine leaders on the current committee. This means we will not be able to apply the Chernoff inequality to bound the number of Byzantine members on a committee. It is unclear whether this is merely a mathematical hurdle or the adversary really has some way to increase its representation on the committee gradually and to go above $n/3$ eventually.

Therefore, we will switch to *external leaders* for reconfiguration. In particular, the successful miner would act as the leader L for reconfiguration and try to elect itself onto the committee. When L's reconfiguration proposal is committed (becomes a consensus decision), reconfiguration is finished, and the system starts processing transactions under the new committee with L being the leader. Note that at this point L becomes a committee member, so the system has seamlessly transitioned back to internal leaders.

By giving all external miners opportunities to become leaders, we introduce a new leader contention problem. It is possible that before L can finish reconfiguration, another miner L' also finds a PoW. Moreover, there may be concurrent internal leaders who are still trying to propose transactions. Solida resolves this type of contention through a Paxos-style leader election [16] with ranks. Only a higher ranked leader can interrupt lower ranked ones. To ensure safety, the higher ranked leader may have to honor the proposals from lower ranked leaders by re-proposing them, if there exists one.

Now the key challenge is to figure out how leaders (internal or external) should be ranked in our protocol. One idea is to rank external leaders by the output hash of their PoW, and stipulate that external leaders are higher ranked than all internal leaders in that configuration. This approach, however, allows the adversary to prevent progress once in a while. If a Byzantine miner submits a "high ranked" PoW but does not drive reconfiguration, the system temporarily stalls until some honest miner finds an even higher ranked PoW. Note that no transactions can be approved in the meantime because internal leaders are all lower ranked than the Byzantine PoW finder. Our solution to this problem is to "expire" stalling external leaders, and give the leader role back to current committee members, so that the committee can resume processing transactions under internal leaders until the next external leader emerges. Crucially, during this process, we must enforce a total order among all leaders, internal or external, to ensure safety. The details are presented in Section 4.

1.2 Overview of the Model and Proof

We adopt the network model of Pass et al. [25]: a bounded message delay of Δ that is known apriori to all participants. Note that this is the standard synchronous network model in distributed computing, though Pass et al. [25] call it "asynchronous networks". What they really meant was that *Bitcoin does not behave like a conventional synchronous protocol.* In a conventional synchronous protocol, participants move forward in synchronized rounds of duration Δ, but Bitcoin does nothing of this sort and has no clear notion of rounds. Our adoption of a synchronous network model may raise a few immediate questions, which we discuss below.

Is the bounded message delay assumption realistic for the Internet? How realistic this assumption is depends a lot on the parameter Δ. With a conservative estimate, say $\Delta = 5$ seconds, and Byzantine fault tolerance, the assumption may be believable. Fault tolerance also helps here. Participants experiencing slow networks and unable to deliver messages within Δ are considered Byzantine. So the assumption we require in Solida is that adversarial participants and "slow" participants collectively control no more than (roughly) 1/3 of the mining power.

More importantly, the bounded network delay assumption seems necessary for all PoW-based protocols. Otherwise, imagine that whenever an honest member finds a PoW, its messages are arbitrarily delayed due to asynchrony. Then, it is as if that the adversary controls 100% of the mining power. In this case, the adversary can create forks of arbitrary lengths in Bitcoin or completely take over the committee in BFT-based blockchains [8, 15, 26]. Pass et al. formalized the above intuition and showed that Bitcoin's security fundamentally relies on the bounded message delay [25]. The larger Δ is, the smaller percentage of Byzantine participants Bitcoin can tolerate; if Δ is unbounded (an asynchronous network), then even an adversary with minuscule mining power can break Bitcoin.

Why do we use PBFT if the network is synchronous? It is well known that there exist protocols that can tolerate $f < n/2$ Byzantine faults in a synchronous network [14, 20, 29]. By requiring $f < n/3$, PBFT preserves safety under asynchrony. But this seems unnecessary given that we assume synchrony. We adopt PBFT (with modifications) because it is an established protocol that provides responsiveness. Most protocols that tolerate $f < n/2$ are synchronous and not responsive, because they advance in rounds of duration Δ. If the actual latency of the network is better than Δ – either because the Δ estimate is too pessimistic, the network is faster in the common case, or the network speed has improved as technology advances – a responsive protocol would offer better performance than a synchronous one. In other words, we opt to use an asynchronous protocol in a synchronous setting, essentially abandoning the strength of the bounded message delay guarantee, in order to run as fast as the network permits. But like all PoW-based blockchains, we rely on synchrony for safety in the worse case. We remark that a recent protocol called XFT [20] seems to be responsive in its steady state while tolerating $f < n/2$ Byzantine faults. An interesting future direction is to explore whether we can combine our reconfiguration techniques and XFT to get the best of both worlds.

Proof outline. If each committee in Solida has no more than $f < n/3$ Byzantine members, then safety mostly follows from traditional Byzantine consensus. To guarantee $f < n/3$, we require that reconfiguration is mostly "fair", i.e., the probability that a newly elected member is honest (Byzantine) is roughly proportional to the mining power of honest (Byzantine)

Table 1 Comparison of consensus protocols in Solida and related designs. Bitcoin and Solida combine committee election and transactions into a single step. Other protocols decouple the two, which we reflect with two separate columns in the table.

Design	Consensus for committee election	Consensus for transactions	Responsive	Defend selfish mining
Bitcoin [22]	Nakamoto		No	No
Bitcoin-NG [9]	Nakamoto	Nakamoto	No	No
PeerCensus [8]	Byzantine*	Byzantine	Yes	No
ByzCoin [15]	Nakamoto/Byzantine	Byzantine	Yes	Yes
Hybrid consensus [26]	Nakamoto	Byzantine	Yes	Yes
Solida	**Byzantine**	**Byzantine**	**Yes**	**Yes**

miners. Additionally, with external leaders driving reconfiguration, we can show independence between reconfiguration events, and can then apply the Chernoff inequality to bound the probability of having $n/3$ Byzantine members on a committee. Thus, the key step of the proof is to show that the Byzantine participants cannot distort their chance of joining the committee by too much. Here, the adversary's main advantage is the network delay for honest participants, which essentially buys extra time for Byzantine miners to find PoWs. But since the network delay is bounded, the Byzantine miners' advantage in finding PoWs can be also bounded.

2 Related Work

A comparison between Solida and related designs. Table 1 compares Solida and related designs. To start, we have Bitcoin that pioneered the direction of permissionless consensus (though not using the term). Bitcoin uses Nakamoto consensus, i.e., PoW and the longest-chain-win rule, and does not separate leader election and transaction processing. There have been numerous follow-up works in the Nakamoto framework. Some "altcoins" simply increases the block creation rate without other major changes to the Bitcoin protocol. A few proposals relax the "chain" design and instead utilize other graph structures to allow for faster block creation rate [32, 19, 31]. Some proposals [28] aim to improve throughput with off-chain transactions.

The key idea of Bitcoin-NG is to decouple transactions and leader election [9]. A miner is elected as a leader if it mines a (key) block in a PoW chain. This miner/leader is then responsible for signing transactions in small batches (microblocks) until a new leader emerges. Since (key) blocks in the PoW chain are small, Bitcoin-NG reduces the likelihood of forks. One can think of each leader in Bitcoin-NG to be a single-member committee. Since a single leader cannot be trusted, any transactions it approves still have to be buried in a Nakamoto PoW chain. Thus, Bitcoin-NG does not improve transaction confirmation time.

To our knowledge, PeerCensus [8] is the first work to envision a committee that approves transactions using PBFT. It makes the observation that Byzantine consensus enables fast transaction confirmation. The description and pseudocode are very high level. Under our best interpretation, it seems that the committee is responsible for reconfiguring itself using Byzantine consensus, but no detail is given on reconfiguration. ByzCoin [15] employs multi-signatures to improve the scalability of PBFT. The reconfiguration algorithm in ByzCoin seems to be left open with multiple options. The description in the paper [15] is quite terse, and can be interpreted in multiple ways. A later blog post [11] suggests adopting hybrid

consensus [26]. If ByzCoin goes this path, then our comparison to hybrid consensus applies to ByzCoin as well. Through private communication, we learned that ByzCoin designers also had in mind a PBFT-style reconfiguration protocol, which seems to involve extra steps not described in the paper. We will have to wait to see a detailed published reconfiguration protocol to compare with ByzCoin properly. Pass and Shi proposed hybrid consensus [26], which we have compared to in detail in Section 1.

Blockchain analysis. While the Bitcoin protocol is quite simple and elegant, proving its security rigorously turns out to be highly nontrivial. The original Bitcoin paper [22] and a few early works [10, 32] considered specific attacks. Garay et al. [12] provided the first comprehensive security analysis for Bitcoin, by modeling Bitcoin as a synchronous protocol running in a synchronous network. Pass et al. [25] refined the analysis after observing that Bitcoin, while relying on a synchronous assumption, does not behave like a conventional synchronous protocol (cf. Section 1.1).

Byzantine Consensus. Consensus is a classic problem in distributed computing. There are a few variants of the consensus problem under Byzantine faults. A theoretical variant is Byzantine agreement [18] in which a fixed set of participants, each with an input value, try to agree on the same value. A more practical variant, which is also more suitable for blockchains, is BFT state machine replication (SMR). In this variant, a fixed set of participants try to agree on a sequence of values that may come from any participant or even external sources. Most BFT SMR protocols follow the PBFT [5] framework.

3 Model

Network. We consider a *permissionless* setting in which participants use their public keys as pseudonyms and can join or leave the system at any time. Participants are connected to each other in a peer-to-peer network with a small diameter. As mentioned, we adopt a synchronous network model: whenever a participant sends a message to another participant, the message is guaranteed to reach the recipient within Δ time. For convenience, we define Δ to be the end-to-end message delay bound. If there are multiple hops from the sender to the recipient, Δ is the time upper bound for traveling all those hops. Our protocol uses computational puzzles, i.e., PoW. Similar to Bitcoin, the difficulty of the puzzle is periodically adjusted so that the expected time to find a PoW stays at D. D should be set significantly larger than Δ according to our analysis in Section 4.5.

Types of participants. Participants in the system are either *honest* or *Byzantine*. Honest participants always follow the prescribed protocol, and are capable of delivering a message to other honest participants within Δ time. Byzantine participants are controlled by a single adversary and can thus coordinate their actions. At any time, Byzantine participants collectively control no more than ρ fraction of the total computation power. We assume the n members in the Genesis committee C_1 are known to all participants, and that the number of Byzantine participants on C_1 is less than $n/3$.

Delayed adaptive adversary. We assume it takes time for the adversary to corrupt a honest participant. It captures the idea that it takes time to bribe a miner or infect a clean host. Most committee-based designs [8, 15, 26] require this assumption. Otherwise, an adversary can easily violate safety by corrupting the entire committee, which may be small compared

to the whole miner population. This is formalized as a delayed adaptive adversary by Pass and Shi [26]. Specifically, we assume that even if the adversary starts corrupting a member as soon as it emerges with a PoW, by the time of corruption, the member would have already left the committee. Whether this assumption holds in practice remains to be examined. Algorand [6] introduced techniques to hide the identities of the committee members, and can thus tolerate instant corruption. Its core technique, however, seems to be tied to proof of stake, which has its own challenges such as the nothing-at-stake problem.

4 The Protocol

4.1 Overview

Solida has a rolling committee running a Byzantine consensus protocol to commit (batches of) transactions and reconfiguration events into a ledger. We say each committed decision fills a *slot* in a linearizable ledger. The protocol should guarantee all honest members commit the same value for each slot (safety) and keeps moving to future slots (liveness). To provide safety and liveness, the number of Byzantine members on each committee must be less than f ($n = 3f + 1$), which will be proved in Section 4.5.

We first describe the protocol outside reconfiguration, i.e., when no miner has found a PoW for the current puzzle. This part further consists of two sub-protocols: *steady state* under a stable leader (Section 4.2), and *view change* to replace the leader (Section 4.3). Members monitor the current leader, and if the leader does not make progress after some time, members support the next leader. The new leader has to learn what values have been proposed, and possibly re-propose those values. This part of the protocol is very similar to PBFT [5] except that the views in our protocol have more structures to support external leaders for *reconfiguration* (Section 4.4).

Each configuration can have multiple lifespans, and each lifespan can have many views. We use consecutive integers to number configurations, lifespans within a configuration, and views within a lifespan. The lifespan number is the number of PoWs (honest) committee members have seen so far in the current configuration. Intuitively, when a new PoW is found, the lifespan of the previous PoW ends. As explained in Section 1.1, our protocol can switch back and forth between internal and external leaders and we must enforce a total order among all leaders. To this end, we will associate each leader with a configuration-lifespan-view tuple (c, e, v), and rank leaders by c first, and then e, and then v. Each (c, e, v) tuple also defines a steady state, in which the leader of that view, denoted as $L(c, e, v)$, proposes values for members to commit. The view tuple and the leader role are managed as follows.

- Upon learning that a reconfiguration event is committed into the ledger, a member M increases its configuration number c by 1, and resets e and v to 0. $L(c, 0, 0)$ is the newly added member by the previous reconfiguration consensus decision.
- Upon receiving a new PoW for the current configuration, a member M increases its lifespan number e by 1, and resets v to 0. M now supports the finder of the new PoW as $L(c, e, 0)$ ($e \geq 1$).
- Upon detecting that the current leader is not making progress (timed out), a member M increases its view number v by 1. $L(c, e, v)$ ($v \geq 1$) is the l-th member on the current committee, by the order they join the committee, where $l = \mathcal{H}(c, e) + v \mod n$, and $\mathcal{H}$ is a random oracle (hash function).

To summarize, the 0-th leader of a lifespan is the newly added member (for $e = 0$) or the new PoW finder ($e \geq 1$). After that, leaders in that lifespan are selected in a round robin

manner, with a pre-defined starting point. We chose a pseudorandom starting point using $\mathcal{H}$ but other choices also work.

Now let us consider whether honest members agree on the identities of the leaders. For a view (c, e, v) with $e = 0$ or $v \geq 1$, the leader identity is derived from a deterministic function on the tuple and the current committee member identities (i.e., previous reconfiguration decisions). As long as the protocol ensures safety so far, members agree on the identities of these leaders. For views of the form $(c, e, 0)$ with $e \geq 1$, members may not agree on the leader's identity if they receive multiple PoWs out of order. This is not a problem for safety because it is similar to having an equivocating leader and a BFT protocol can tolerate leader equivocation. However, contending leaders in a view may prevent progress. Luckily, members will time out and move on to views (c, e, v) with $v \geq 1$. Unique leaders in those views will ensure liveness.

Crucially, our protocol forbids pipelining to simplify reconfiguration: a member M sends messages for slot s only if it has committed all slots up to $s - 1$. Let $\mathcal{M}(c, e, v, s)$ be the set of members currently in view (c, e, v) working on slot s. Each honest member M can only belong to one such set at any point in time.

We use digital signatures even under a stable leader, i.e., we do not use PBFT's MAC optimization [5]. The MAC optimization is not effective for a cryptocurrency since there are already hundreds of digital signatures to verify in each block/slot. We use $\langle x \rangle_M$ to denote a message x that carries a signature from member M. A message can be signed in layers, e.g., $\langle \langle x \rangle_M \rangle_{M'}$. When the context is clear, we omit the signer and simply write $\langle x \rangle$ or $\langle \langle x \rangle \rangle$.

When we say a member "broadcasts" a message, we mean the member sends the message to all current committee members including itself.

4.2 Steady State

- (**Propose**) The leader L picks a batch of valid transactions $\{\mathsf{tx}\}$ and then broadcasts $\{\mathsf{tx}\}$ and $\langle \mathsf{propose}, c, e, v, s, h \rangle_L$ where h is the hash digest of $\{\mathsf{tx}\}$.
  After receiving $\{\mathsf{tx}\}$ and $\langle \mathsf{propose}, c, e, v, s, h \rangle_L$, a member $M \in \mathcal{M}(c, e, v, s)$ checks
  - $L = L(c, e, v)$ and L has not sent a different proposal,
  - s is a *fresh* slot, i.e., no value could have been committed in slot s (details in Section 4.4),
  - $\{\mathsf{tx}\}$ is a set of valid transactions whose digest is h.
- (**Prepare**) If all the above checks pass, M broadcasts $\langle \mathsf{prepare}, c, e, v, s, h \rangle$.
  After receiving $2f + 1$ matching prepare messages, a member $M \in \mathcal{M}(c, e, v, s)$ *accepts* the proposal (represented by its digest h), and concatenates the $2f + 1$ matching prepare messages into an *accept certificate* $\mathcal{A}$.
- (**Commit**) Upon accepting h, M broadcasts $\langle \mathsf{commit}, c, e, v, s, h \rangle$.
  After receiving $2f + 1$ matching commit messages, a member $M \in \mathcal{M}(c, e, v, s)$ commits $\{\mathsf{tx}\}$ into slot s, and concatenates the $2f + 1$ matching commit messages into a *commit certificate* $\mathcal{C}$.

  The above is standard PBFT. We now add an extra propagation step to the steady state.

- (**Notify**) Upon committing h, M sends $\langle \langle \mathsf{notify}, c, e, v, s, h \rangle, \mathcal{C} \rangle_M$ to all other members to notify them about the decision. M also starts propagating this decision on the peer-to-peer network to miners, users and merchants, etc. M then moves to slot $s + 1$.
  Upon receiving a notify message like the above, a member commits h, sends and propagates its own notify message if it has not already done so, and then moves to slot $s + 1$.

4.3 View Change

A Byzantine leader cannot violate safety, but it can prevent progress by simply not proposing. Thus, in PBFT, every honest member M monitors the current leader, and if no new slot is committed after some time, M abandons the current leader and supports the next leader. Since we have assumed a worst-case message delay of Δ, it is natural to set the timeout based on Δ. The concrete timeout values (4Δ and 8Δ) will be justified in the proof of Theorem 2.

- (**View-change**) Whenever a member M moves to a new slot s in a steady state, it starts a timer T. If T reaches 4Δ and M still has not committed slot s, then M abandons the current leader and broadcasts $\langle \mathsf{view\text{-}change}, c, e, v \rangle$.
  Upon receiving $2f+1$ matching view-change messages for (c, e, v), if a member M is not in a view higher than (c, e, v), it forwards the $2f+1$ view-change messages to the new leader $L' = L(c, e, v+1)$. After that, if M does not receive a new-view message from L' within 2Δ, then M abandons L' and broadcasts $\langle \mathsf{view\text{-}change}, c, e, v+1 \rangle$.
- (**New-view**) Upon receiving $2f+1$ matching view-change messages for (c, e, v), the new leader $L' = L(c, e, v+1)$ concatenates them into a view-change certificate $\mathcal{V}$, broadcasts $\langle \mathsf{new\text{-}view}, c, e, v+1, \mathcal{V} \rangle_{L'}$, and enters view $(c, e, v+1)$.
  Upon receiving a $\langle \mathsf{new\text{-}view}, c, e, v, \mathcal{V} \rangle_{L'}$ message, if a member M is not in a view higher than (c, e, v), it enters view (c, e, v) and starts a timer T. If T reaches 8Δ and still no new slot is committed, then M abandons L' and broadcasts $\langle \mathsf{view\text{-}change}, c, e, v \rangle$.
- (**Status**) Upon entering a new view (c, e, v), M sends $\langle \langle \mathsf{status}, c, e, v, s-1, h, s, h' \rangle, \mathcal{C}, \mathcal{A} \rangle$ to the new leader $L' = L(c, e, v)$. In the above message, h is the value committed in slot $s-1$ and $\mathcal{C}$ is the corresponding commit certificate; h' is the value accepted by M in slot s and $\mathcal{A}$ is the corresponding accept certificate ($h' = \mathcal{A} = \perp$ if M has not accepted any value for slot s). We call the inner part of the message (i.e., excluding $\mathcal{C}, \mathcal{A}$) its *header*.
  Upon receiving $2f+1$ status, L' concatenates the $2f+1$ status headers to form a status certificate $\mathcal{S}$. L' then picks a status message that reports the highest last-committed slot s^*; if there is a tie, L' picks the one that reports the highest ranked accepted value in slot s^*+1 . Let the two certificates in this message be $\mathcal{C}^*$ and $\mathcal{A}^*$ ($\mathcal{A}^*$ might be $\perp$).
- (**Re-propose**) The new leader L' broadcasts $\langle \mathsf{repropose}, c, e, v, s^*+1, h', \mathcal{S}, \mathcal{C}^*, \mathcal{A}^* \rangle_{L'}$. In the above message, s^* should be the highest last-committed slot reported in $\mathcal{S}$. h' should match the value in $\mathcal{A}^*$ if $\mathcal{A}^* \neq \perp$; If $\mathcal{A}^* = \perp$, then L' can choose h' freely. [2]
  The repropose message serves two purposes. First, $\mathcal{C}^*$ allows everyone to commit slot s^*. Second, it re-proposes h' for slot s^*+1, and carries a proof $(\mathcal{S}, \mathcal{A}^*)$ that h' is safe for slot s^*+1. The repropose message is invalid if any of the aforementioned conditions is violated: s^* is not the highest committed slot, $\mathcal{C}$ is not for slot s^*, $\mathcal{A}$ is not for the highest ranked accepted value for s^*+1, or h' is not the value certified by $\mathcal{A}$.
  Upon receiving a valid repropose message, a member M commits slot s^* if it has not already; M then executes the prepare/commit/notify steps as in the steady state for slot s^*+1, and marks all slots $> s^*+1$ *fresh* for view (c, e, v). At this point, the system transitions into a new steady state.

A practical implementation of our protocol can piggyback status messages on view-change messages and repropose messages on new-view messages to save two steps. We choose to separate them in the above description because the reconfiguration sub-protocol can reuse the status step.

[2] Strictly speaking, in this case, L' is proposing a new value rather than re-proposing. But we keep the message name as repropose for convenience. A similar situation exists in reconfiguration.

4.4 Reconfiguration

PoW to prevent Sybil attacks. In order to join the committee, a new member must show a PoW, i.e., a solution to a computational puzzle based on a random oracle $\mathcal{H}$. The details of the puzzle will be discussed later. For now, it is only important to know that each configuration defines a new puzzle. Let the puzzle for configuration c be $\mathsf{puzzle}(c)$. Solutions to this puzzle are in the form of $(\mathsf{pk}, \mathsf{nonce})$. pk is a miner's public key. A miner keeps trying different values of nonce until it finds a valid solution such that $\mathcal{H}(\mathsf{puzzle}(c), \mathsf{pk}, \mathsf{nonce})$ is smaller than some threshold. The threshold is also called the *difficulty* and is periodically adjusted to keep the expected reconfiguration interval at D.

Upon finding a PoW, the miner tries to join the committee by driving a reconfiguration consensus as an external leader. To do so, it first broadcasts the PoW to committee members.

- (**New-lifespan**) Upon finding a PoW, the miner broadcasts it to the committee members. Upon receiving a new valid PoW for the current configuration (that M has not seen before), M forwards the PoW to other members, enters view $(c, e+1, 0)$, sets the PoW finder as its 0-th leader $L' = L(c, e+1, 0)$ of the new lifespan, and starts a timer T. If T reaches 8Δ and still no new slot is committed, then M abandons L' and broadcasts $\langle \mathsf{view\text{-}change}, c, e+1, 0 \rangle$.
- (**Status**) Upon entering a new lifespan, M sends L' $\langle \langle \mathsf{status}, c, e, 0, s-1, h, s, h' \rangle, \mathcal{C}, \mathcal{A} \rangle$, where $s, h, \mathcal{C}, \mathcal{A}$ are defined the same way as in the view-change sub-protocol.
  Upon receiving $2f+1$ status, L' constructs s^*, $\mathcal{S}$, $\mathcal{C}^*$ and $\mathcal{A}^*$ as in view-change.
- (**Re-propose and Reconfigure**) Let h^* be the committed value in the highest committed slot s^* among $\mathcal{S}$. Let h' be the highest ranked accepted value (could be $\perp$) into slot s^*+1 among $\mathcal{S}$. Note that h^* and h' are certified by $\mathcal{S}, \mathcal{C}^*$ and $\mathcal{A}^*$. Now depending on whether h^* and h' are transactions or reconfiguration events, the new external leader L' takes different actions.
  - If h^* is a reconfiguration event to configuration $c+1$, then L' simply broadcasts $\mathcal{C}$ and terminates. Terminating means L' gives up its endeavor to join the committee (it is too late and some other leader has already finished reconfiguration) and starts working on $\mathsf{puzzle}(c+1)$.
  - If h^* is a batch of transactions, and h' is a reconfiguration event to configuration $c+1$, then L' broadcasts $\langle \mathsf{repropose}, c, e, 0, s^*+1, h', \mathcal{S}, \mathcal{C}^*, \mathcal{A}^* \rangle_{L'}$ and then terminates.
  - If h^* is a batch of transactions, and $h' = \perp$, then L' tries to drive the reconfiguration consensus into slot s^*+1 by broadcasting $\langle \mathsf{repropose}, c, e, 0, s^*+1, h, \mathcal{S}, \mathcal{C}^*, \mathcal{A}^* \rangle_{L'}$ where h is a reconfiguration event that lets L' join the committee. If this proposal becomes committed, then starting from slot s^*+2, the system reconfigures to the next configuration and L' joins the committee replacing the oldest member.
  - If h^* and h' are both batches of transactions, then L' first re-proposes h' for slot s^*+1 by broadcasting $\langle \mathsf{repropose}, c, e, 0, s^*+1, h', \mathcal{S}, \mathcal{C}^*, \mathcal{A}^* \rangle_{L'}$. After that, L' tries to drive a reconfiguration consensus into slot s^*+2 by broadcasting $\langle \mathsf{propose}, c, e, 0, s^*+2, h \rangle_{L'}$ where h is a reconfiguration event that lets L' join the committee. If this proposal becomes committed, then starting from slot s^*+3, the system reconfigures to the next configuration and L' joins the committee replacing the oldest member.

In the latter three cases, members react to the $\mathsf{repropose}$ message in the same way as in view-change: check its validity and execute prepare/commit/notify. Note that the reconfiguration proposal in the last case is a steady state $\mathsf{propose}$ message instead of a $\mathsf{repropose}$ message, and it does not need to carry certificates as proofs. This is because after members commit the re-proposed value into slot s^*+1, the system enters a special

steady state under leader L'. Slot $s^* + 2$ will be marked as a *fresh* slot in this steady state. This steady state is special in the sense that, if L' is not faulty and is not interrupted by another PoW finder, members will experience a configuration change, but the leader remains unchanged as we have defined $L(c+1, 0, 0)$ to be the newly added member. After reconfiguration, it becomes a normal steady state in view $(c+1, 0, 0)$.

The puzzle and defense to selfish mining. We now discuss what exactly the puzzle is. A natural idea is to make the puzzle be the previous reconfiguration decision. However, this allows *withholding attacks* similar to selfish mining [10]. To favor the adversary, we assume that if the adversary and an honest miner both find PoWs, the adversary wins the contention and joins the committee. Then, when the adversary finds a PoW first, its best strategy is to temporarily withhold it. At this point, the adversary already knows the next puzzle, which is its own reconfiguration proposal. Meanwhile, honest miners are wasting their mining power on the current puzzle, not knowing that they are doomed to lose to the adversary when they eventually find a PoW. This gives the adversary a head start on the next puzzle. Its chance of taking the next seat on the committee will hence be much higher than its fair share of mining power. The authors of ByzCoin suggested an idea to thwart the above attack: let the puzzle include $2f + 1$ committee member signatures. This way, a PoW finder cannot determine the puzzle on its own and thus gains nothing from withholding its PoW. For concreteness, we let $\mathsf{puzzle}(c+1)$ be *any* set of $f + 1$ notify headers for the last reconfiguration decision $\langle \mathsf{notify}, c, e, v, s, h \rangle$. This ensures the following:

▶ **Lemma 1.** *The adversary learns a puzzle at most* 2Δ *time earlier than honest miners.*

Proof. For the adversary to learn the puzzle, at least one honest committee member must broadcast its notify message, which will cause all other honest members to broadcast their notify messages within Δ time. After another Δ time, all honest miners receive enough notify messages to learn the puzzle. ◀

4.5 Safety and Liveness

We first prove safety and liveness *assuming* each committee has no more than $f < n/3$ Byzantine members. The proof of the following theorem (given in the appendix) mostly follows PBFT. Essentially, the biggest change we made to PBFT at an algorithmic level is from a round robin leader schedule to a potential interleaving between internal and external leaders. But crucially, we still have a total order among leaders to ensure safety and liveness.

▶ **Theorem 2.** *The protocol achieves safety and liveness if each committee has no more than* $f < n/3$ *Byzantine members.*

Next, we show $f < n/3$ indeed holds. Let ρ be the adversary's ratio of mining power. Honest miners thus collective control $1 - \rho$ of the total network mining power. Ideally, we would hope that reconfiguration is a "fair game", i.e., the adversary taking the next committee seat is with probability ρ, and an honest miner takes the next seat with probability $1 - \rho$. This is indeed simply assumed to be the case in PeerCensus [8] and ByzCoin [15]. However, we show that due to network latency, it is as if that the adversary has an *effective mining power* $\rho' = 1 - (1 - \rho)e^{-(2\rho+8)\Delta/D} > \rho$. (Recall that D is the expected time for the entire network to find a PoW.) We need $D \gg \Delta$ so that ρ' is not too much larger than ρ.

▶ **Theorem 3.** *Assuming* $f < n/3$ *holds for* C_1*, then* $f < n/3$ *holds for each subsequent committee except for a probability exponentially small in* n *if* $\rho' = 1-(1-\rho)e^{-(2\rho+8)\Delta/D} < 1/3$.

Proof. The increase in the adversary's effective mining power comes from three sources. First, as Lemma 1 shows, the adversary may learn a puzzle up to 2Δ earlier than honest miners. Second, if the adversary finds a PoW while an honest external leader is reconfiguring, which can take up to 8Δ time, the adversary can interrupt the honest leader and win the reconfiguration race. Lastly, two honest leaders may interrupt each other if they find PoWs within 8Δ time apart, while the adversary-controlled miners can coordinate their actions. Therefore, an honest miner wins a reconfiguration if all three events below happen:

- (event X) the adversary finds no PoW in its 2Δ head start,
- (event Y) some honest miner finds a PoW earlier than the adversary given X, and
- (event Z) no miner, honest or adversarial, finds a PoW in the next 8Δ time given Y.

PoW mining is a *memory-less* process modeled by a Poisson distribution: the probability for a miner to find k PoWs in a period is $p(k,\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$, where λ is the expected number of PoWs to be found in this period. Recall that the expected time for all miners combined to find a PoW is D. In the 2Δ head start, the adversary expects to find $\lambda_X = 2\Delta\rho/D$ PoWs. Thus, $\Pr(X) = p(0,\lambda_X) = e^{-2\Delta\rho/D}$. Similarly, $\Pr(Z) = e^{-8\Delta/D}$. Event Y is a "fair race" between the adversary and honest miners, so $\Pr(Y) = 1-\rho$. The memory-less nature of PoW also means that the above events X, Y and Z are independent. Thus, the probability that an honest miner wins a reconfiguration is

$$\Pr(X \cap Y \cap Z) = \Pr(X)\Pr(Y)\Pr(Z) = (1-\rho)e^{-(2\rho+8)\Delta/D} := 1-\rho'.$$

We define the above probability to be $1-\rho'$, which can be thought of as the honest miners' effective mining power once we take message delays into account. ρ' is then the adversary's effective mining power. This matches our intuition: as $D \to \infty$, the adversary's advantage from message delay decreases, and $(1-\rho') \to (1-\rho)$.

Conditioned on all committees up to C_i having no more than $n/3$ Byzantine members, the adversary wins each reconfiguration race with probability at most ρ', independent of the results of other reconfiguration races. We can then use the Chernoff inequality to bound the probability of Byzantine members exceeding $n/3$ on any committee up to C_{i+1}. Let Q denote the number of Byzantine members on a committee. We have $E(Q) := \mu = \rho' n$. $\Pr(Q \geq (1+\delta)\mu) \leq e^{\frac{-\delta\mu\log(1+\delta)}{2}}$ due to the Chernoff bound. Select $\delta = \frac{1}{3\rho'} - 1$, we have $\Pr(Q \geq n/3) \leq e^{\frac{n(1-3\rho')\log(3\rho')}{6}}$. If $\rho' < 1/3$, then $(1-3\rho')\log(3\rho') < 0$, and the above probability is exponentially small in n as required. ◀

Concrete committee size. We calculate the required committee size under typical values. We can calculate ρ' from ρ given the value of Δ/D. If Δ is 5 seconds and D is 10 minutes, then $\Delta/D = 1/120$. Then, for a desired security level, a simple calculation of binomial distribution yields the required committee size n. The results are listed in Table 2. A security parameter k means there is a 2^{-k} probability that, after a reconfiguration, the adversary controls more than $n/3$ seats on the committee. $k = 25$ or $k = 30$ should be sufficient in practice. Assuming we reconfigure every 10 minutes, 2^{25} reconfigurations take more than 600 years and 2^{30} reconfigurations take more than 20,000 years. With $\rho' = 25\%$, the required committee size is around 1000.

It is worth noting that as ρ' approaches 33%, the required committee size increases rapidly and the committee-based approach becomes impractical. This gap between theory and practice is expected. Similarly, Bitcoin, in theory, can tolerate an adversarial with up to 50% mining power. But the recommended "six confirmation" is calculated assuming a 10% adversary and a rather low security level $k \approx 10$. If the adversary really controls close to 50% mining power, then thousands of confirmations would be required for each block.

Table 2 The required committee size under different adversarial miner ratios ρ and desired security levels k, assuming $\Delta/D = 1/120$.

k		20	25	30	35	40
$\rho = 14\%$	$\rho' = 20\%$	232	298	367	439	508
$\rho = 20\%$	$\rho' = 25\%$	649	841	1036	1231	1423
$\rho = 23\%$	$\rho' = 28\%$	1657	2149	2644	3142	3580
$\rho = 25\%$	$\rho' = 30\%$	4366	5650	6949	8248	9256

4.6 Towards Pipelining

One limitation of the above Solida protocol is that it forbids pipelining, i.e., members are not allowed to work on multiple slots in parallel. The interaction between pipelining and reconfiguration is non-trivial, and to the best of our knowledge, has not been worked out in BFT protocols. In particular, when a slot is being worked on, the exact configuration (i.e., identities of committee members) for that slot would not be known if previous slots have not been committed. In this subsection, we briefly describe how we plan to support pipelining. We hope to present and implement a detailed pipelined protocol in a future version.

The key idea is to let each proposal carry a digest of its "predecessor proposal". Since the predecessor proposal also carries the digest of its own predecessor proposal, each proposal is tied to its entire chain of "ancestor proposals". A non-leader member M sends messages for a proposal for slot s only if M has accepted its predecessor proposal in slot $s-1$. When a member accepts a proposal, it also accepts all its ancestors proposals; likewise, when a member commits a proposal, it also commits all its ancestors proposals.

During a view-change or a reconfiguration, a member reports in its status message the highest ranked proposal chain that it has accepted (with an accept certificate). The new leader L' then re-proposes the highest ranked proposal chain it hears from members. Note that in either step, we prioritize the highest ranked proposal chain, not the longest proposal chain. If the highest ranked proposal chain contains a reconfiguration event in its tail (highest numbered slot), then L' re-proposes this proposal chain and terminates. Otherwise, L' re-proposes the proposal chain and then start making its own proposals. If L' is a PoW finder, then its proposal will be a reconfiguration proposal. If its proposal goes through, L' joins the committee and the system enters a new configuration. The first leader in the new configuration is the newly added member L', and the system transitions back to a steady state with L' now serving as an internal leader.

5 Implementation and Evaluation

Implementation details. We implement the non-pipelined version of the Solida consensus protocol in Python and measure its performance. We implement our 6-phase PBFT-style protocol in which committee members agree on a proposal digest from a successful miner. For digital signatures, we use ECDSA with the prime192v2 curve and the Charm crypto library [2]. We test Solida on 16 m4.16xlarge machines on Amazon EC2. The CPUs are Intel Xeon E5-2686 v4 @ 2.3 GHz. Each machine has 64 cores, giving a total of 1024 cores. We use 1 CPU core for each committee member, testing committee size up to 1000. To emulate Internet latency and bandwidth, we add a delay of 0.1 second (unless otherwise stated) to each message and throttle the bandwidth of each committee member to the values in Figure 1.

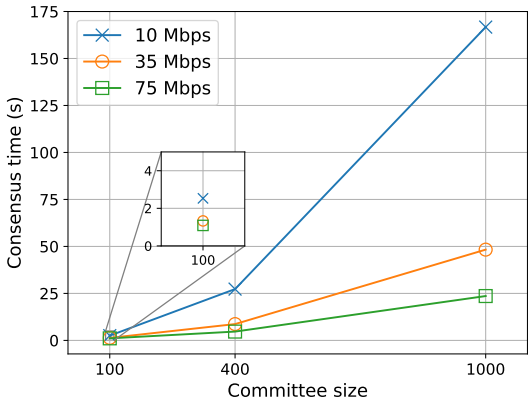


Figure 1 Time to achieve reconfiguration consensus with different committee sizes and network bandwidth.

Experimental results. The main result we report is the time it takes for committee members to commit a reconfiguration consensus decision (committing a decision in the steady state takes strictly less time). We report the consensus time from the leader's perspective, i.e., the time between when the leader sends its PoW and when the leader receives the first Notify message, which is the earliest point at which the leader knows its proposal is committed. Figure 1 presents the time to reach a reconfiguration consensus decision (averaged over 10 experiments) under different committee sizes n and bandwidth limits. As shown in the zoomed-in part, with a small committee of 100 members, a decision takes 1 to 3 seconds depending on the throughput. The bottleneck here is network latency. Recall that there are 8 sequential messages from the leader's perspective, each adding 0.1s latency. With larger committees of 400 and 1000 members, network bandwidth becomes the bottleneck. As expected, consensus time is roughly quadratic in committee size and inversely proportional to network bandwidth. With moderate network bandwidth, consensus time is manageable even with 1000 committee members: 48.3s for 35 Mbps and 23.6s for 75 Mbps. To study the effect of network latency, we rerun some experiments with a 0.5s latency (results not shown). In each experiment, the consensus time increases by about 3s, which is as expected given the 8 sequential messages and a 0.4s latency increase for each.

6 Conclusion and Future Work

This work presents Solida, a blockchain protocol based on PoW-augmented reconfigurable Byzantine consensus. We have presented a detailed protocol, rigorously proved safety and liveness under the (seemingly necessary) bounded message delay model, and provided a prototype implementation and evaluation results. Besides the pipelined protocol, interesting future directions include designing and rigorously analyzing incentives for Solida, and extending Solida to tolerate an adversary with up to 50% mining power.

Acknowledgement. We thank Eleftherios Kokoris-Kogias for clarifying the ByzCoin protocol and other helpful discussions. We thank the anonymous reviewers for their valuable suggestions.

References

1. Ittai Abraham and Dahlia Malkhi. BVP: Byzantine vertical paxos, 2016.
2. Joseph A Akinyele, Christina Garman, Ian Miers, Matthew W Pagano, Michael Rushanan, Matthew Green, and Aviel D Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013.
3. Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 2014.
4. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. pages 514–532. Springer, 2001.
5. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186. USENIX Association, 1999.
6. Jing Chen and Silvio Micali. Algorand: The efficient and democratic ledger, 2016. URL: `https://arxiv.org/pdf/1607.01341.pdf`.
7. CoinDesk. Average number of transactions per block, accessed Aug, 2016. URL: `https://www.coindesk.com/data/bitcoin-number-transactions-per-block/`.
8. Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, page 13. ACM, 2016.
9. Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, 2016.
10. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International Conference on Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.
11. Bryan Ford. Untangling mining incentives in bitcoin and byzcoin, 2016. URL: `http://bford.github.io/2016/10/25/mining/`.
12. Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.
13. Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.
14. Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *J. Comput. Syst. Sci.*, 75(2):91–112, 2009.
15. Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium*, pages 279–296. USENIX Association, 2016.
16. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
17. Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313. ACM, 2009.
18. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

19 Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

20 Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 485–500. USENIX Association, 2016.

21 Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

22 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

23 Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 305–320. IEEE, 2016.

24 Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

25 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673. Springer International Publishing, 2017.

26 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. 2016. Cryptology ePrint Archive, Report 2016/917.

27 Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324. ACM, 2017.

28 Joseph Poon and Thaddeus Dryja. The Bitcoin lightning network: Scalable off-chain instant payments. Technical Report., 2015. URL: `https://lightning.network`.

29 Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. Practical synchronous byzantine consensus. Cryptology ePrint Archive, Report 2017/307, 2017. URL: `http://eprint.iacr.org/2017/307`.

30 Rodrigo Rodrigues, Barbara Liskov, Kathryn Chen, Moses Liskov, and David Schultz. Automatic reconfiguration for large-scale reliable storage systems. *IEEE Transactions on Dependable and Secure Computing*, 9(2):145–158, 2012.

31 Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. SPECTRE: A fast and scalable cryptocurrency protocol, 2016.

32 Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

A Proofs

Proof of Theorem 2. We first consider safety. Let M be the member to commit slot s in the lowest ranked view. Say M commits value h and it does so in view (c, e, v). We need to show that no other value $h' \neq h$ can be committed into slot s in that view and all future views. In fact, we will show by induction that there cannot exist an accept certificate $\mathcal{A}'$ for h' in that view and all future views, which means no honest member would have accepted h', let alone committing it. Since M committed in view (c, e, v), there must be $2f + 1$ members that have accepted h in that view, and $2f + 1$ members that have sent $\langle \mathsf{prepare}, c, e, v, h \rangle$. For the base case, suppose for contradiction that $\mathcal{A}'$ exists for h' in view (c, e, v). Then, $2f + 1$ members must send $\langle \mathsf{prepare}, c, e, v, h' \rangle$. This means at least one honest member has sent prepare messages for two different proposals in view (c, e, v), which is a contradiction. For the inductive step, suppose that no accept certificate exists for h' from view (c, e, v) up to (excluding) view (c', e', v'). Since M has committed slot s, then at least $2f + 1$ members

must have committed slot $s-1$ (they send prepare for slot s only after committing slot $s-1$). So in the status certificate $\mathcal{S}$ of a repropose message for view (c', e', v'), the largest last-committed slot $s^* \geq s-1$. Therefore, slot s is either already committed to h, or has to be re-proposed. If slot s needs to be re-proposed, which means $s^* = s-1$, then $\mathcal{S}$ must contain at least one status header reporting that h has been accepted into slot s with a rank no lower than (c, e, v). Due to the inductive hypothesis, no accept certificate can exist for h' with a rank equal to or higher than (c, e, v). h is, therefore, the unique highest ranked accepted value for slot s, and it is the only value that can be legally re-proposed in view (c', e', v'). Hence, no accept certificate for h' can exist in view (c', e', v'), completing the proof.

Next, we consider liveness. According to the protocol, if a Byzantine leader does not commit any slot for too long, it will be replaced shortly. But here is an another liveness attack that a Byzantine leader can perform in the cryptocurrency setting. The Byzantine leader can simply construct transactions that transfer funds between its own accounts and keep proposing/committing these transactions. It is impossible for honest members to detect such an attack. Fortunately, we have shown that honest miners win at least 2/3 of the reconfiguration races, and after a reconfiguration, the newly added member becomes the leader. So 2/3 of the time, an honest leader is in control to provide liveness. It remains to check that an honest leader will not be replaced until the next lifespan. It is sufficient to show that an honest leader will never be accused by an honest member. This is guaranteed by our timeout values. First, observe that the notify step ensures two honest members commit a slot at most Δ time apart. In the steady state, each member gives the leader 4Δ time to commit each slot: one Δ to account for the possibility that other members come to this slot (i.e., finish the previous slot) Δ time later, and three Δ for the three phases in the steady state. At the beginning of a new view or a new lifespan, each member gives the leader 8Δ time, because the status and repropose steps take extra 4Δ time. The last case to consider is when a member abandons a future leader because it does not send new-view when it should. Before accusing a future leader, a member allows 2Δ time after forwarding it a view-change certificate, which is sufficient for an honest leader to broadcast new-view. ◀

Remark. Garay et al. [12] laid out three desired properties for blockchains: common prefix, chain quality, and chain growth. We note that these three properties are more applicable to Bitcoin-style blockchains that do not satisfy traditional safety and liveness. Common prefix is strictly weaker than safety. Chain growth is the Nakamoto consensus counterpart of liveness. If chain quality is interpreted as "the ratio of honest committee members" in the BFT-based approach, then we have proved it in Theorem 3. If it is instead interpreted as "the ratio of slots committed under honest leaders", then it is not very meaningful. We cannot prevent Byzantine leaders from making progress really fast, thereby decreasing the ratio of slots committed under honest leaders, but that does not hurt honest leaders' ability to commit slots at their own pace in their own views.

B Comparison to ByzCoin Evaluation Results

We would like to explain an apparent contradiction between our experimental results and those of ByzCoin [15]. Under the same parameter settings (0.1s latency, 35 Mbps bandwidth and 100 committee members), our implementation of PBFT takes only 1.3 seconds per consensus decision, while ByzCoin concludes that PBFT has unacceptable performance. As a result, ByzCoin suggests sending (and aggregating) consensus messages in a tree rooted at

the leader. On a closer look, our results and theirs actually corroborate each other. ByzCoin finds PBFT performance to be unacceptable only when the block is large. With a small block size, their results confirm that PBFT is very fast and indeed takes about 1s per consensus decision. Therefore, the unacceptable performance they see in PBFT solely results from the leader broadcasting the entire batch of transactions $\{\mathsf{tx}\}$. It is indeed important to reduce the burden on the leader by gossiping $\{\mathsf{tx}\}$ among committee members. But using a tree as the communication graph is not Byzantine fault tolerant. A Byzantine node in the tree can make its entire subtree unreachable. Instead, $\{\mathsf{tx}\}$ should be sent through the peer-to-peer network much like in Bitcoin. The other major technique ByzCoin proposed, the Schnorr multi-signature, is also not Byzantine fault tolerant. Schnorr signature has a "commitment" step before the actual signing step. A Byzantine signer may participate in the commitment step but refuse to sign in the signing step. The leader then has to initiate a new instance of Schnorr multi-signature up to f times, which may be even slower than using normal signatures. Therefore, the results reported in ByzCoin are only for the best case where there is no adversary. These results are still relevant if we believe most of the time during the protocol, there is no adversary. Alternatively, we may use multi-signature schemes that do not have the commitment step and are thus Byzantine fault tolerant [4]. But these schemes are much less efficient, and it remains interesting future work to study whether they improve efficiency in practice.

Efficient and Modular Consensus-Free Reconfiguration for Fault-Tolerant Storage*†

Eduardo Alchieri[1], Alysson Bessani[2], Fabíola Greve[3], and Joni da Silva Fraga[4]

1 **University of Brasília, Brazil**
alchieri@unb.br
2 **LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal**
anbessani@ciencias.ulisboa.pt
3 **Federal University of Bahia, Brazil**
fabiola@dcc.ufba.br
4 **Federal University of Santa Catarina, Brazil**
fraga@das.ufsc.br

Abstract

Quorum systems are useful tools for implementing consistent and available storage in the presence of failures. These systems usually comprise of a static set of servers that provide a fault-tolerant read/write register accessed by a set of clients. We consider a dynamic variant of these systems and propose FreeStore, a set of fault-tolerant protocols that emulates a register in dynamic asynchronous systems in which processes are able to join/leave the set of servers during the execution. These protocols use a new abstraction called *view generators*, that captures the agreement requirements of reconfiguration and can be implemented in different system models with different properties. Particularly interesting, we present a reconfiguration protocol that is *modular, efficient, consensus-free* and *loosely coupled* with read/write protocols. An analysis and an experimental evaluation show that the proposed protocols improve the overall system performance when compared with previous solutions.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Distributed Systems, Reconfiguration, Fault-Tolerant Quorum Systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.26

1 Introduction

Quorum systems [13] are a fundamental abstraction to ensure consistency and availability of data stored in replicated servers. Apart from their use as building blocks of synchronization protocols (e.g., consensus [7, 19]), quorum-based protocols for read/write (r/w) register implementation are appealing due to their scalability and fault tolerance: the r/w operations do not need to be executed in all servers, but only in a quorum of them. The consistency of the stored data is ensured by the intersection between any two quorums.

Quorum systems were initially studied in static environments, where servers are not allowed to join or leave the system during execution [4,13]. This approach is not adequate for

* This work was partially supported by CNPq (Brazil) through project FREESTORE (Universal 457272/2014-7) and by FCT (Portugal) through projects LaSIGE (UID/CEC/00408/2013) and IRCoC (PTDC/EEI-SCR/6970/2014).

† A full version of the paper is available at [3], https://arxiv.org/abs/1607.05344.

long lived systems since given a sufficient amount of time, there might be more faulty servers than the threshold tolerated, affecting the system correctness. Beyond that, this approach does not allow a system administrator to deploy new machines (to deal with increasing workloads) or replace old ones at runtime. Moreover, these protocols can not be used in many systems where, by their very nature, the set of processes that compose the system may change during its execution (e.g., MANETs and P2P overlays).

Reconfiguration is the process of changing the set of nodes that comprise the system. Previous solutions proposed for reconfigurable storage, by implementing dynamic quorum systems, rely on consensus for reconfigurations in a way that processes agree on the set of servers (view) supporting the storage [14, 20]. Although adequate, since the problem of changing views resembles an agreement problem, this approach is not the most efficient or appropriate. Besides the costs of running the protocol, consensus is known to not be solvable in asynchronous environments [11]. Moreover, atomic shared memory emulation can be implemented in static asynchronous systems without requiring consensus [4].

DynaStore [2] was the first protocol that implement dynamic atomic storage without relying on consensus for reconfigurations. These reconfigurations may occur at any time and generate a graph of views from which it is possible to identify a sequence of views in which clients need to execute their r/w operations (Figure 1, left). Unfortunately, DynaStore has two serious drawbacks: (1) its reconfiguration and r/w protocols are strongly tied and (2) its performance is significantly worse than consensus-based solutions in synchronous executions, which are the norm. The first issue is particularly important since it means that DynaStore r/w protocols (as well as other consensus-based works like RAMBO [14]) explicitly deal with reconfigurations and are quite different from static r/w protocols (e.g., ABD protocol [4]).

More recently, the SmartMerge [17] and SpSn [12] protocols improved DynaStore by separating the reconfiguration and r/w protocols. A common framework was proposed for their implementations [21]. Although these approaches make it easy to adapt other static register implementations to dynamic environments, they do not fully decouple the execution of r/w and reconfiguration protocols, since before they execute each r/w operation, it is necessary to access some reconfiguration abstraction to check for updates in the system. These abstractions are implemented by a set of static single-writer multi-reader (SWMR) registers. We show by experiments (Section 7) that this design decision significantly reduces the system performance, even in periods without reconfigurations.

In this paper we present FreeStore, a set of algorithms for implementing fault-tolerant atomic [18] and wait-free [15] storage that allows the reconfiguration of the set of servers at runtime. FreeStore is composed by two types of protocols: (1) r/w protocols and (2) the reconfiguration protocol. Read/write protocols can be adapted from static register implementations (e.g., the classical ABD [4], as used in this paper). The reconfiguration protocol – our main contribution – is used to change the set of replicas supporting the storage. The key innovation of the FreeStore reconfiguration protocol is the use of *view generators*, a new abstraction that captures the agreement requirements of reconfiguration protocols. We provide two implementations of view generators – based on consensus and consensus-free – and compare how efficiently they can be used to solve reconfiguration.

FreeStore improves the state of the art in at least three aspects: *modularity*, *efficiency* and *simplicity*. The modularity of the proposed protocols is twofold: it separates the reconfiguration from the r/w protocols, allowing other static storage protocols (e.g., [8, 10]) to be adapted to our dynamic model, and introduces the notion of view generators, capturing the agreement requirements of a reconfiguration protocol. Moreover, this modularity is designed in a way that it does not impact r/w operations in periods without reconfigurations. In

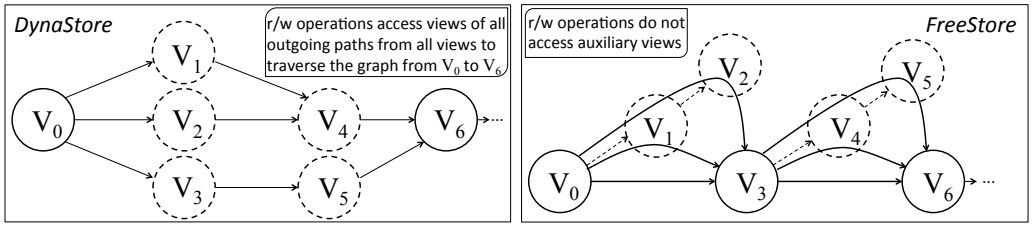


Figure 1 View convergence strategies of DynaStore [2] (left) and FreeStore (right). Dotted circles represent the auxiliary/non-established views that the system may experience during a reconfiguration. Solid circles represent the installed/established views process must converge to.

terms of performance, both the r/w and reconfiguration protocols of FreeStore require less communication steps than their counterparts from previous works, either consensus-based [14] or consensus-free [2,12,17]. In particular, FreeStore' consensus-free reconfiguration requires less communication steps than other consensus-based reconfiguration protocols in the best case, matching the intuition that a consensus-free approach should be faster than another that rely on consensus. Finally, the consensus-free variant of FreeStore introduces a novel reconfiguration strategy that reduces the number of intermediary installed views that a process must traverse to reach the current installed view with all requested updates (see Figure 1, right). This strategy is arguably easier to understand than the one used in previous works, shedding more light on the study of consensus-free reconfiguration protocols.

In summary, this paper makes the following contributions:

1. It introduces the notion of *view generators*, an abstraction that captures the agreement requirements of a storage reconfiguration protocol and provides two implementations: the consensus-based *perfect view generator* and the consensus-free *live view generator*.
2. It shows that *safe* and *live* dynamic fault-tolerant atomic storage can be implemented using the proposed view generators and discusses the tradeoffs between them.
3. It presents FreeStore, the first dynamic atomic storage system in which the reconfiguration protocol *(a)* can be configured to be consensus-free or consensus-based and *(b)* is fully decoupled from the r/w protocols, increasing the system performance and making it easy to adapt other static register implementation to dynamic environments.
4. It presents some experiments and a detailed comparison of FreeStore with previous protocols [2,12,14,17], showing that FreeStore is faster (i.e., requires less communication steps) than these protocols.

2 Preliminary Definitions

2.1 System Model

We consider a fully-connected distributed system composed by a universe of processes U, that can be divided in two non-overlapping subsets: an infinite set of servers $\Pi = \{1, 2, ...\}$; and an infinite set of clients $C = \{c_1, c_2, ...\}$. Clients access the storage system provided by a subset of the servers (a *view*) by executing read and write operations (*r/w* for short). Each process (client or server) of the system has a unique identifier. Servers and clients are prone to *crash failures*. Crashed processes are said to be *faulty*. A process that is not faulty is said to be *correct*. Moreover, there are *reliable channels* connecting all pairs of processes [6].

We assume an *asynchronous distributed system* in which there are no bounds on message transmission delays and processing times. However, each server has access to a local clock

used to trigger reconfigurations. These clocks are not synchronized and do not have any bounds on their drifts, being nothing more than counters that keep increasing. Besides that, there is a real-time clock not accessed by processes, used in definitions and proofs.

2.2 Dynamic Storage Properties

During a dynamic system execution, a sequence of views is installed to account for replicas joining and leaving. In the following we describe some preliminary definitions and the properties satisfied by FreeStore.

Updates. We define $update = \{+, -\} \times \Pi$, where the tuple $\langle +, i\rangle$ (resp. $\langle -, i\rangle$) indicates that server i asked to join (resp. leave) the system. A *reconfiguration* procedure takes into account updates to define a new system's configuration, which is represented by a *view*.

Views. A view v is composed by a set of *updates* (represented by $v.entries$) and its associated membership (represented by $v.members$). Consequently, $v.members = \{i \in \Pi \colon \langle +, i\rangle \in v.entries \wedge \langle -, i\rangle \notin v.entries\}$. To simplify the notation, we sometimes use $i \in v$ and $|v|$ meaning $i \in v.members$ and $|v.members|$, respectively. Notice that a server i can join and leave the system only once, but this condition can be relaxed in practice if we add an epoch number on each reconfiguration request.

We say a view v is *installed* in the system if some correct server $i \in v$ considers v as its *current view* and answers client r/w operations on this view. When v is installed, we say that the previous installed view (before v) was *uninstalled* from the system. At any time t, we define $V(t)$ to be the *most up-to-date view* (see definition below) installed in the system. We consider that $V(t)$ remains *active* from the time it is installed in the system until *all correct servers* of another most up-to-date view $V(t'), t' > t$, installs $V(t')$.

Comparing views. We compare two views v_1 and v_2 by comparing their *entries*. We use the notation $v_1 \subset v_2$ and $v_1 = v_2$ as an abbreviation for $v_1.entries \subset v_2.entries$ and $v_1.entries = v_2.entries$, respectively. If $v_1 \subset v_2$, then v_2 is *more up-to-date* than v_1.

Bootstrapping. We assume a non-empty initial view $V(0)$ known to all processes. At system startup, each server $i \in V(0)$ receives an initial view $v_0 = \{\langle +, j\rangle \colon j \in V(0)\}$.

Views vs. r/w operations. At any time t, r/w operations are executed only in $V(t)$. When a server i asks to *join* the system, these operations are disabled on it until an *enable operations* event occurs. After that, i remains able to process r/w operations until it asks to leave the system, which will happen after the occurence of a *disable operations* event.

▶ **Definition 1** (FreeStore properties)**.** FreeStore satisfies the following properties:

- **Storage Safety**: The r/w protocols satisfy the safety properties of an atomic r/w register [18].
- **Storage Liveness**: Every r/w operation executed by a correct client eventually completes.
- **Reconfiguration – Join Safety**: If a server j installs a view v such that $i \in v$, then server i has invoked the *join* operation or i is member of the initial view.

- **Reconfiguration – Leave Safety**: If a server j installs a view v such that $i \notin v \wedge (\exists v' : i \in v' \wedge v' \subset v)$, then server i has invoked the *leave* operation.[1]
- **Reconfiguration – Join Liveness**: Eventually, the *enable operations* event occurs at every correct server that has invoked a *join* operation.
- **Reconfiguration – Leave Liveness**: Eventually, the *disable operations* event occurs at every correct server that has invoked a *leave* operation.

2.3 Additional Assumptions for Dynamic Storage

Dynamic fault-tolerant storage protocols [2, 12, 14, 17, 20] require the following additional assumptions to deal with the dynamism.

▶ **Assumption 2** (Fault threshold). *For each view v, we denote $v.f$ as the number of faults tolerated in v and assume that $v.f \leq \lfloor \frac{|v.members|-1}{2} \rfloor$.*

▶ **Assumption 3** (Quorum size). *For each view v, we assume quorums of size $v.q = \lceil \frac{|v.members|+1}{2} \rceil$.*

These two assumptions are a direct adaptation of the optimal resilience for fault-tolerant quorum systems [4] to account for multiple views and only need to hold for views present in the generated view sequences (see Section 3).

▶ **Assumption 4** (Gentle leaves). *A correct server $i \in V(t)$ that asks to leave the system at time t remains in the system until it knows that a more up-to-date view $V(t'), t' > t, i \notin V(t')$ is installed in the system.*

This assumption ensures that a *correct* leaving server will participate in the reconfiguration protocol that installs the new view without itself, i.e., it cannot leave the system before a new view accounting for its removal is installed. Other dynamic systems require similar assumption: departing replicas need to stay available to transfer their state to arriving replicas. Notice the fault threshold accounts for faults while the view is being reconfigured.

▶ **Assumption 5** (Finite reconfigurations). *The number of updates requested in an execution is finite.*

As in other dynamic storage systems, this assumption is fundamental to ensure r/w operations termination. It ensures that a client will restart phases of a r/w operation a finite number of times, and thus, eventually complete its operation. In practice, updates could be infinite as long as each r/w is concurrent with a finite number of reconfigurations.

3 View Generators

View generators are distributed oracles used by servers to generate sequences of new views for system reconfiguration. This module aims to capture the agreement requirements of reconfiguration algorithms. In order to be general enough to be used for implementing consensus-free algorithms, such requirements are reflected in the sequence of generated views, and not directly on the views. This happens because, as described in previous works [1, 2, 12, 14, 17], the key issue with reconfiguration protocols is to ensure that the sequence of (possibly conflicting) views generated during a reconfiguration procedure will converge to a single view with all requested view updates.

[1] We can relax this property and adapt our protocol (Section 4) to allow that other process issues the leave operation on behalf of a crashed process.

For each view v, each server $i \in v$ associates a view generator $\mathcal{G}_i^v$ with v in order to generate a succeeding sequence of views. Server i interacts with a view generator through two primitives: (1) $\mathcal{G}_i^v.gen\_view(seq)$, called by i to propose a new view sequence seq to update v; and (2) $\mathcal{G}_i^v.new\_view(seq')$, a callback invoked by the view generator to inform i that a new view sequence seq' was generated for succeeding v. An important remark about this interface is that there is no one-to-one relationship between *gen_view* and *new_view*: a server i may not call the first but receive several upcalls on the latter for updating the *same view* (e.g., due to reconfigurations started by other servers). However, if i calls $\mathcal{G}_i^v.gen\_view(seq)$, it will eventually receive at least one upcall to $\mathcal{G}_i^v.new\_view(seq')$.

Similarly to other distributed oracles (e.g., failure detectors [7]), view generators can implement these operations in different ways, according to the different environments they are designed to operate (e.g., synchronous or asynchronous systems). However, in this paper we consider view generators satisfying the following properties.

▶ **Definition 6** (View Generators)**.** A generator $\mathcal{G}_i^v$ (associated with v in server i) satisfy the following properties:

- **Accuracy**: we consider two variants of this property:
  - **Strong Accuracy**: for any $i, j \in v$, if i receives an upcall $\mathcal{G}_i^v.new\_view(seq_i)$ and j receives an upcall $\mathcal{G}_j^v.new\_view(seq_j)$, then $seq_i = seq_j$.
  - **Weak Accuracy**: for any $i, j \in v$, if i receives an upcall $\mathcal{G}_i^v.new\_view(seq_i)$ and j receives an upcall $\mathcal{G}_j^v.new\_view(seq_j)$, then either $seq_i \subseteq seq_j$ or $seq_j \subset seq_i$.
- **Non-triviality**: for any upcall $\mathcal{G}_i^v.new\_view(seq_i)$, $\forall w \in seq_i$, $v \subset w$.
- **Termination**: if a correct server $i \in v$ calls $\mathcal{G}_i^v.gen\_view(seq)$, then eventually it will receive an upcall $\mathcal{G}_i^v.new\_view(seq_i)$.

Accuracy and Non-triviality are safety properties while Termination is related to the liveness of view generation. Furthermore, the Non-triviality property ensures that generated sequences contain only updated views.

Using the two variants of accuracy we can define two types of view generators: $\mathcal{P}$, the *perfect view generator*, that satisfies *Strong Accuracy* and $\mathcal{L}$, the *live view generator*, that only satisfies *Weak Accuracy*. Our implementation of $\mathcal{P}$ requires consensus, while $\mathcal{L}$ can be implemented without such strong synchronization primitive.

3.1 Perfect View Generators – $\mathcal{P}$

Perfect view generators ensure that a single sequence of views is generated at all servers using the generators. Our implementation for $\mathcal{P}$ (Algorithm 1) uses a deterministic Paxos-like consensus protocol [19] that assumes a partially synchronous system model [9]. Under the Paxos framework, any server of v can be a proposer (calling $Paxos^v.propose(seq)$), but all servers of v are acceptors and learners (they receive an upcall $Paxos^v.learn(seq')$, even if they did not propose anything). We also assume that once a value is learned for a Paxos instance associated with v, this value is locked and no other value will be learned. Notice that a server only proposes a sequence if the new views are strict extensions of v, ensuring Non-triviality. Termination and Strong Accuracy properties comes directly from the Agreement and Termination properties of the underlying consensus algorithm [7, 19].

3.2 Live View Generators – $\mathcal{L}$

Algorithm 2 presents an implementation for *Live view generators* ($\mathcal{L}$). Our algorithm does not require a consensus building block, being thus implementable in asynchronous systems.

Algorithm 1 $\mathcal{P}$ associated with v - server $i \in v$.

upon $\mathcal{G}_i^v.gen\_view(seq)$
1) **if** $\forall w \in seq : v \subset w$ **then** $Paxos^v.propose(seq)$ //starts a consensus in v
upon $Paxos^v.learn(seq')$ // when decides by seq' ...
2) $\mathcal{G}_i^v.new\_view(seq')$ // ... inform this to process.

Algorithm 2 $\mathcal{L}$ associated with v - server $i \in v$.

functions: Auxiliary functions
$most\_updated(seq) \equiv w$: $(w \in seq) \wedge (\nexists w' \in seq$: $w \subset w')$
variables: Sets used in the protocol
$\texttt{SEQ}^v \leftarrow \emptyset$ // proposed view sequence
$\texttt{LCSEQ}^v \leftarrow \emptyset$ // last converged view sequence known
procedure $\mathcal{G}_i^v.gen\_view(seq)$
1) **if** $\texttt{SEQ}^v = \emptyset \wedge \forall w \in seq : v \subset w$ **then**
2) $\texttt{SEQ}^v \leftarrow seq$
3) $\forall j \in v, send\langle$SEQ-VIEW$, \texttt{SEQ}^v\rangle$ to j
upon receipt of $\langle$SEQ-VIEW$, seq\rangle$ from j
4) **if** $\exists w \in seq$: $w \notin \texttt{SEQ}^v$ **then**
5) **if** $\exists w, w'$: $w \in seq \wedge w' \in \texttt{SEQ}^v \wedge w \not\subset w' \wedge w' \not\subset w$ **then**
6) $w \leftarrow most\_updated(\texttt{SEQ}^v)$
7) $w' \leftarrow most\_updated(seq)$
8) $\texttt{SEQ}^v \leftarrow \texttt{LCSEQ}^v \cup \{w.entries \cup w'.entries\}$
9) **else**
10) $\texttt{SEQ}^v \leftarrow \texttt{SEQ}^v \cup seq$
11) $\forall k \in v, send\langle$SEQ-VIEW$, \texttt{SEQ}^v\rangle$ to k
upon receipt of $\langle$SEQ-VIEW$, \texttt{SEQ}^v\rangle$ from $v.q$ servers in v
12) $\texttt{LCSEQ}^v \leftarrow \texttt{SEQ}^v$
13) $\forall k \in v, send\langle$SEQ-CONV$, \texttt{SEQ}^v\rangle$ to k
upon receipt of $\langle$SEQ-CONV$, seq'\rangle$ from $v.q$ servers in v
14) $\mathcal{G}_i^v.new\_view(seq')$

On the other hand, it can generate different sequences in different servers for updating the same view. We bound such divergence by exploiting Assumptions 2 and 3, which ensure that any quorum of the system will intersect in at least one correct server, making any generated sequence for updating v be contained in any other posterior sequence generated for v (Weak Accuracy). Furthermore, the servers keep updating their proposals until a quorum of them converges to a sequence containing all proposed views (or combinations of them), possibly generating some intermediate sequences before this convergence.

To generate a new sequence of views, a server $i \in v$ uses an auxiliary function *most_updated* to get the most up-to-date view in a sequence of views (i.e., the view that is not contained in any other view of the sequence). Moreover, each server keeps two local variables: $\texttt{SEQ}^v$ – the last view sequence proposed by the server – and $\texttt{LCSEQ}^v$ – the last sequence this server converged. When server $i \in v$ starts its view generator, it first verifies (1) if it already made a proposal for updating v and (2) if the sequence being proposed contains only updated views (line 1). If these two conditions are met, it sends its proposal to the servers in v (lines 2-3).

Different servers of v may propose sequences containing different views and therefore these views need to be organized in a sequence. When a server $i \in v$ receives a proposal for a view sequence from $j \in v$, it verifies (line 4) if this proposal contains some view it did not know yet (notice a server could receive this message even if its view generator was not yet initialized, i.e., $\texttt{SEQ}^v = \emptyset$). If this is the case, i updates its proposal ($\texttt{SEQ}^v$) according to two mutually exclusive cases:

- Case 1 [There are *conflicting views* in the sequence proposed by i and the sequence received from j (lines 5-8)]: In this case i creates a new sequence containing the last converged sequence ($\texttt{LCSEQ}^v$) and a new view with the union of the two most up-to-date conflicting views. This maintains the containment relationship between any two generated view sequences.
- Case 2 [The sequence proposed by i and the received sequence *can be composed in a new sequence* (lines 9-10)]: The new sequence is the union of the two known sequences.

In both cases, a new proposal containing the new sequence is disseminated (line 11). When i receives the same proposal from a quorum of servers in v, it *converges* to $\texttt{SEQ}^v$ and stores it in $\texttt{LCSEQ}^v$, informing other servers of v about it (lines 12-13). When i knows that a quorum of servers of v converged to some sequence seq', it *generates* seq' (line 14).

Correctness (full proof in [3]). The algorithm ensures that if a quorum of servers converged to a sequence seq' (lines 12-13), then *(1)* such sequence will be generated (line 14) and *(2)* any posterior sequence generated will contain seq' (lines 8 and 10), ensuring Weak Accuracy. This holds due to the quorum intersection: at least one correct server needs to participate in the generation of both sequences and this server applies the rules in Cases 1 and 2 to ensure that sequences satisfy the containment relationship. The Termination property is ensured by the fact that *(1)* each server makes at most one initial proposal (lines 1-3); *(2)* servers keep updating their proposals until a quorum agree on some proposal; and *(3)* there is always a quorum of correct servers in v.

4 FreeStore Reconfiguration

A server running FreeStore reconfiguration algorithm uses a view generator associated with its current view cv to process one or more reconfiguration requests (joins and leaves) that will lead the system from cv to a new view w. Algorithm 3 describes how a server i executes reconfigurations. In the following sections we first describe how view generators are started and then we proceed to discuss the behavior of this algorithm when started with either $\mathcal{L}$ (Section 4.2) or $\mathcal{P}$ (Section 4.3).

4.1 View Generator Initialization

Algorithm 3 describes how a server i processes reconfiguration requests and starts a view generator associated with its current view cv (lines 1-7). A server j that wants to join the system needs first to find the current view cv and then to execute the *join* operation (lines 1-2), sending a tuple $\langle +, j \rangle$ to the members of cv. Servers leaving the system do something similar, through the *leave* operation (lines 3-4). When i receives a reconfiguration request from j, it verifies if the requesting server is using the same view as itself; if this is not the case, i replies its current view to j (omitted from the algorithm for brevity). If they are using the same view and i did not execute the requested reconfiguration before, it stores this request in its set of pending updates `RECV` and sends an acknowledgment to j (lines 5-6).

For the sake of performance, a local *timer* has been defined in order to periodically process the updates requested in a view, that is, the next system reconfiguration. A server $i \in cv$ starts a reconfiguration for cv when its timer expires and i has some pending reconfiguration requests (otherwise, the timer is renewed). The view generator is started with a sequence containing a single view representing the current view plus the pending updates (line 7).

Algorithm 3 FREESTORE reconfiguration - server i.

functions: Auxiliary functions
  $least\_updated(seq) \equiv w\colon (w \in seq) \wedge (\nexists w' \in seq\colon w' \subset w)$
variables: Sets used in the protocol
  $cv \leftarrow v_0$ // the system current view known by i
  $\texttt{RECV} \leftarrow \emptyset$ // set of received updates
procedure *join*()
1) $\forall j \in cv, send\langle$RECONFIG$, \langle +, i\rangle, cv\rangle$ to j
2) **wait** for $\langle$REC-CONFIRM$\rangle$ replies from $cv.q$ servers in cv
procedure *leave*()
3) $\forall j \in cv, send\langle$RECONFIG$, \langle -, i\rangle, cv\rangle$ to j
4) **wait** for $\langle$REC-CONFIRM$\rangle$ replies from $cv.q$ servers in cv
upon receipt of $\langle$RECONFIG$, \langle *, j\rangle, cv\rangle$ from j and $\langle *, j\rangle \notin cv$
5) $\texttt{RECV} \leftarrow \texttt{RECV} \cup \{\langle *, j\rangle\}$
6) $send\langle$REC-CONFIRM$\rangle$ to j
upon ($timeout$ for cv) $\wedge(\texttt{RECV} \neq \emptyset)$
7) $\mathcal{G}_i^{cv}.gen\_view(\{cv \cup \texttt{RECV}\})$
upon $\mathcal{G}_i^{ov}.new\_view(seq)$ //$\mathcal{G}$ generates a new sequence of views to update ov (usually $ov = cv$)
8) $w \leftarrow least\_updated(seq)$ //the next view in the sequence seq
9) $R\text{-}multicast(\{j\colon j \in ov \vee j \in w\}, \langle$INSTALL-SEQ$, w, seq, ov\rangle)$
upon $R\text{-}delivery(\{j\colon j \in ov \vee j \in w\}, \langle$INSTALL-SEQ$, w, seq, ov\rangle)$
10) **if** $i \in ov$ **then** //i is member of the previous view in the sequence
11) **if** $cv \subset w$ **then** *stop the execution of r/w operations* //if w is more up-to-date than cv stop r/w
12) $\forall j \in w, send\langle$STATE-UPDATE$, \langle val, ts\rangle, \texttt{RECV}\rangle$ to j //i sends its state to servers in the next view
13) **if** $cv \subset w$ **then** //w is more up-to-date than cv and the system will be reconfigured from cv to w
14) **if** $i \in w$ **then** //if i is in the new view...
15) **wait** for $\langle$STATE-UPDATE$, *, *\rangle$ messages from $ov.q$ servers in ov //... it updates...
16) $\langle val, ts\rangle \leftarrow \langle val_h, ts_h\rangle$, pair with highest timestamp among the ones received //... its state...
17) $\texttt{RECV} \leftarrow \texttt{RECV} \cup \{$update requests from STATE-UPDATE messages$\} \setminus w.entries$
18) $cv \leftarrow w$ //... and its current view to w
19) **if** $i \notin ov$ **then** *enable operations* //i is joining the system
20) $\forall j \in ov \setminus cv, send\langle$VIEW-UPDATED$, cv\rangle$ to j //inform servers in $ov \setminus cv$ that they can leave
21) **if** $(\exists w' \in seq\colon cv \subset w')$ **then** //there are views more up-to-date than cv in seq...
22) $seq' \leftarrow \{w' \in seq\colon cv \subset w'\}$ //... gather these views...
23) $\mathcal{G}_i^{cv}.gen\_view(seq')$ //... and propose them (going back to Algorithm 2)
24) **else**
25) *resume the execution of r/w operations* in $cv = w$ and start a timer for cv //w is installed
26) **else** //i is leaving the system
27) *disable operations*
28) **wait** for $\langle$VIEW-UPDATED$, w\rangle$ messages from $w.q$ servers in w and then **halt**

4.2 Reconfiguration using $\mathcal{L}$

Overview. Given a sequence $seq : v_1 \rightarrow \ldots \rightarrow v_k \rightarrow w$ generated by *live view generators* ($\mathcal{L}$) for updating a view v, Algorithm 3 ensures that only the last view w will be installed in the system. The other k *auxiliary views* are used only as intermediate steps for installing w.

The use of auxiliary views is fundamental to ensure that no write operation executed in any of the views of seq (in case they are installed in some server) "is lost" by the reconfiguration processing. This is done through the execution of a *"chain of reads"* in which servers of v transfer their state to servers of v_1, which transfer their state to servers of v_2 and so on until servers of w have the most up-to-date state. To avoid consistency problems, r/w operations are disabled during each of these state transfers.

It is important to remark that since we do not use consensus, a subsequence $seq' : v_1 \rightarrow \ldots \rightarrow v_j, j \leq k$, of seq may lead to the installation of v_j in some servers that did not know seq and that these servers may execute r/w operations in this view. However, the algorithm ensures these servers eventually will reconfigure from v_j to the most up-to-date view w.

Protocol. Algorithm 3 (lines 8-28) presents the core of the FreeStore reconfiguration protocol. This algorithm uses an auxiliary function *least_updated* to obtain the least updated view in a sequence of views (i.e., the one that is contained in all other views of the sequence) and two local variables: the aforementioned `RECV` – used to store pending reconfiguration requests – and cv – the current view of the server (initially v_0).

When the view generator associated with some view ov (we use ov instead of cv because view generators associated with *old views*, $ov \subseteq cv$, still active can generate new sequences) reports the generation of a sequence of views seq, the server obtains the least updated view w of seq and proposes this sequence for updating ov through an INSTALL-SEQ message sent to the servers of both, ov and w. Once view generators could generate different sequences at different processes, we employ a *reliable multicast* primitive [7] to ensure all correct servers in ov and w process seq (lines 8-9). This primitive can be efficiently implemented in asynchronous systems with a message retransmission at the receivers before its delivery [7].

The current view is updated through the execution of lines 10-28. First, if the server is a member of the view being updated ov, it must send its state (usually, the register's value and timestamp) to the servers in the new view to be installed (lines 10-12). However, if the server will be updating its current view (i.e., if w is more up-to-date than cv) it first needs to stop executing client r/w operations (line 11) and enqueue these operations to be executed when the most up-to-date view in the sequence is installed (line 25, as discussed below). A server will update its current view only if the least updated view w of the proposed sequence is more up-to-date than its current view cv (line 13). If this is the case, either (1) server i will be in the next view (lines 14-25) or (2) not (line 26-28).

- Case 1 [i will be in the next view (it may be joining the system)]: If the server will be in the next view w, it first waits for the state from a quorum of servers from the previous view ov and then defines the current value and timestamp of the register (lines 15-16), similarly to what is done in the 1st phase of a read operation (see Section 5). After ensuring that its state is updated, the server updates cv to w and, if it is joining the system, it enables the processing of r/w operations (which will be queued until line 25 is executed). Furthermore, the server informs leaving servers that its current view was updated (line 20). The final step of the reconfiguration procedure is the verification if the new view will be installed or not (in case it is an auxiliary view). If $cv = w$ is not the most up-to-date view of the generated sequence seq, a new sequence with more up-to-date views than cv will be proposed for updating it (lines 21-23). Otherwise, cv is installed and server i resumes processing r/w operations (lines 24-25).
- Case 2 [i is leaving the system]: A server leaving the system only halts after ensuring that the view w to which it sent its state was started in a quorum of servers (lines 27-28).

Although the algorithm restarts itself in line 23, it eventually terminates since the number of reconfiguration requests is finite (Assumption 5). Furthermore, since all sequences generated by $\mathcal{L}$ can be composed in an unique sequence, when the reconfiguration terminates in all correct servers, they will have installed the same view with all requested updates.

Correctness (full proof in [3]). Algorithm 3 ensures that *an unique sequence of views is installed in the system* due to the following: (1) if a view w is installed, any previously installed view $w' \subset w$ is uninstalled and will not be installed anymore (lines 11, 18 and 25); consequently, (2) no view more up-to-date than w is installed and the installed views form an unique sequence. By Assumption 5, the reconfiguration procedure always terminate by installing a final view v_{final}. The *Storage Safety* and *Storage Liveness* properties are discussed

in Section 5. The remaining properties of Definition 1 are ensured as follows. *Reconfiguration Join/Leave Safety* are trivially ensured by the fact that only a server i sends the update request $\langle +, i\rangle/\langle -, i\rangle$ (lines 1-4). *Reconfiguration Join/Leave Liveness* are ensured by the fact that if an update request from a server i is stored in `RECV` of a quorum, then it is processed in the next reconfiguration since a quorum with the same proposal is required to generate a view sequence (Algorithm 2). Moreover, update requests received during a reconfiguration are sent to the next view (lines 12-17).

4.3 Reconfiguration using $\mathcal{P}$

If $\mathcal{P}$ is used with the FreeStore reconfiguration protocol (Algorithm 3), all generators will generate the same sequence of views (Strong Accuracy) with a single view w. This will lead the system directly from its current view cv to w (lines 22-23 will never be executed).

5 Read and Write Protocols

This section discusses how a static storage protocol can be adapted to dynamic systems by using FreeStore reconfigurations. Since reconfigurations are decoupled from r/w protocols, they are very similar to their static versions. In a nutshell, there are two main requirements for using our reconfiguration protocol. First, each process (client or server) needs to handle a current view variable cv that stores the most up-to-date view it knows. All r/w protocol messages carry cv and clients update it as soon as they discover that there is a more recent view installed in the system. The servers reject any operation issued to an old view, and reply their current view to the issuing client, which updates its cv. The client restarts the phase of the operation it is executing if it receives an updated view. The second requirement is that, before accessing the system, a client must obtain the system's current view. This can be done by making servers put the current view in a directory service [1] or making the client flood the network asking for it. Notice this is an intrinsic problem for any dynamic system and similar assumptions are required in previous reconfiguration protocols [2, 12, 14, 17, 20].

In this paper we extend the classical ABD algorithm [4] for supporting multiple writers and to work with the FreeStore reconfiguration. In the following we highlight the main aspects of these protocols (complete algorithms are found in [3]). The protocols to read and write from the dynamic distributed storage work in phases. Each phase corresponds to an access to a quorum of servers in cv. The *read protocol* works as follows:

- 1st Phase: a reader client requests a set of tuples $\langle val, ts\rangle$ from a quorum of servers in cv (val is the value the server stores and ts is its associated timestamp) and selects the one with highest timestamp $\langle val_h, ts_h\rangle$; the operation ends and returns val_h if all returned pairs are equal, which happens in executions without write contention or failures;
- 2nd Phase: otherwise, the reader client performs an additional *write-back phase* in the system and waits for confirmations from a quorum of servers in cv before returning val_h.

The *write protocol* works in a similar way:

- 1st Phase: a writer client obtains a set of timestamps from a quorum of servers in cv and chooses the highest, ts_h; the timestamp to be written ts is defined by incrementing ts_h and concatenating the writer id in its lowest bits;
- 2nd Phase: the writer sends a tuple $\langle val, ts\rangle$ to the servers of cv, writing val with timestamp ts, and waits for confirmations from a quorum.

The proposed decoupling of r/w protocols from reconfigurations (1) makes it easy to adapt other static fault-tolerant register implementations to dynamic environments (as long as they work in phases), (2) makes it possible to run r/w in parallel with reconfigurations,

and (3) does not impact the performance of a r/w in periods without reconfigurations. This happens because, differently from previous approaches [2, 12, 14, 17] where a r/w may access multiple views, in FreeStore a client executes these operations only in the most up-to-date installed view, which is received directly from the servers. To enable this, it is required that r/w operations to be blocked during the state transfer between views.

Correctness (full proof in [3]). The above algorithms implement atomic storage in the absence of reconfigurations [4]. When integrated with FreeStore, they also have to satisfy the *Storage Safety* and *Storage Liveness* properties of Definition 1. We start by discussing *Storage Liveness*, which follows directly from the termination of reconfigurations. As discussed before, clients' r/w operations are concluded only if they access a quorum of servers using the same view as the client, otherwise they are restarted. By Assumption 5, the system reconfiguration always terminate by installing some final view. Consequently, a client will restart phases of its operations a finite number of times until this final view is installed in a quorum. *Storage Safety* comes directly from three facts: (1) a r/w operation can only be executed in a single *installed* view (i.e., all servers in the quorum of the operation have the same installed current view), (2) all installed views form a unique sequence , and (3) any operation executed in a view v will still be "in effect" when a more up-to-date view w is installed. More precisely, assume $\langle val, ts \rangle$ is the last value read or written in v, and thus it was stored in $v.q$ servers from v. During a reconfiguration from v to w, r/w operations are disabled until all servers of v send $\langle val, ts \rangle$ to the servers of w (line 12), which terminate the reconfiguration only after receiving the state from a quorum of servers of v. Consequently, all servers who reconfigure to w will have $\langle val, ts \rangle$ as its register's state (lines 15-16). This ensures that any operation executed in w will not miss the operations executed in v.

6 Discussion

6.1 DynaStore vs. FreeStore

This section discusses some differences between DynaStore [2] and FreeStore. Although our focus is on comparing our approach with DynaStore, we also comment the relationship between these protocols and SpSn [12] and SmartMerge [17].

Convergence Strategy. In DynaStore, the reconfiguration process generate a graph of views through which it is possible to identify a sequence of *established* views (see Figure 1, left). A view that is not established works as an *auxiliary view*, that must be accessed during r/w operations. For any established view v, the maximum number of views that can immediately succeed it is $|v|$, and new views representing the combinations of these views could also be generated. In contrast, reconfigurations in FreeStore install only a single sequence of views (see Figure 1, right). In this case, different generated sequences of views are organized in an unique sequence of installed views. For each installed view v, our implementation of $\mathcal{L}$ bounds the number of generated view sequences to $|v| - v.q + 1$. SpSn and SmartMerge also install a sequence of ordered configurations (views).

Liveness. In DynaStore, a process executes a leave and halts the system. However, for any time t, there is a bound on the number of processes that can leave the system without compromising liveness. Let $F(t)$ be the set of processes that crashed until time t and $J(t)$ (resp. $L(t)$) the set of pending joins (resp. leaves) at t. The liveness condition of DynaStore states that fewer than $|V(t)|/2$ processes out of $V(t) \cup J(t)$ should be in $F(t) \cup L(t)$ [2]. In

Table 1 Communication steps of r/w operations.

Operation	*Updated View*				*Outdated View*			
	DS	SM	SpSn	FS	DS	SM	SpSn	FS
Read	12	8	14	2/4	19	16	22	4/6
Write	12	8	18	4	19	16	26	6

contrast, in FreeStore a process that executes a leave should wait for the installation of the updated view (without itself). If this restriction is not respected, the leaving server is considered faulty. This approach specifies a bound on the number of leaves for FreeStore: fewer than $|V(t)|/2$ processes out of $V(t)$ should be in $F(t) \cup L(t)$. SpSn does not specify a liveness condition and SmartMerge uses policies to prevent the generation of unsafe views, which in practice restricts the number of allowed leaves.

Normal Case Execution. In DynaStore, reconfigurations generate a graph of views through which it is possible to identify a sequence of *established* views. A view that is not established works as an *auxiliary view*, but must be accessed during r/w operations. For each view (established or not), DynaStore associates a *weak snapshot object* that is used to store updates. For any view v, its weak snapshot object wso_v is supported by a set S_v of $|v|$ static SWMR registers, i.e., one register for each member of v. During a r/w on v, wso_v must be accessed twice to verify if some update was executed on v. Each access to wso_v comprises two reads in each register of S_v. Thus, to execute a r/w on v, it is necessary a total of $4|v|$ (4 sequential, $|v|$ parallel) quorum accesses, with two communication steps each. SpSn and SmartMerge use a similar approach: before executing each r/w, a set of $|v|$ SWMR registers must be accessed to check for updates. In contrast, the overhead introduced by FreeStore on a r/w is the local verification if the client view is equal to the current view of the servers.

R/W and Reconfiguration Concurrency. A r/w operation needs to traverse the graph of views generated by DynaStore to find a view where it is safe to be executed (the most up-to-date established view). During the transversal, each edge of the graph must be accessed in order to verify if there is a more up-to-date view. For each view v, it is necessary to access its weak snapshot object, which requires $2|v|$ (2 sequential, $|v|$ parallel) quorum accesses. While the system is converging to some established view, the r/w operation does not terminate. Similarly, in SpSn and SmartMerge, a r/w concurrent with a reconfiguration also needs to access intermediary views to find the most up-to-date installed view. FreeStore follows a different approach that ensures r/w operations are directed to the most up-to-date installed view v, without accessing any auxiliary view. However, after some sequence of views for updating v is obtained, the r/w operations are stopped and can only terminate after the installation of the most up-to-date view of this sequence. Therefore, in this sense our approach resembles view-synchronous group communication [5, 6].

6.2 Performance of Read/Write Operations

Table 1 shows the number of communication steps demanded to execute a r/w operation with DynaStore (DS) [2], SmartMerge (SM) [17], SpSn [12] and FreeStore (FS), for a process that handles an updated or outdated view. A r/w operation in FreeStore requires at most a third of the number of communication steps required to execute it in DynaStore or SpSn and at most half of the steps required in SmartMerge. More important, it matches the performance of static protocols in the absence of reconfigurations.

Table 2 Communication steps for reconfiguration.

Consensus-free	*Best Case*	*Worst Case*	*Consensus-based*	*Best Case*	*Worst Case*
DynaStore [2]	23	$18\|v\| + 5$	RAMBO [14]	7	7
SmartMerge [17]	11	$6\|v\| + 5$	FreeStore ($\mathcal{P}$)	5	5
SpSn [12]	14	$8\|v\| + 6$			
FreeStore ($\mathcal{L}$)	4	$7\|v\| - 2v.q - 1$			

6.3 Consensus-free vs. Consensus-based Reconfiguration

Table 2 shows the number of communication steps required to process a reconfiguration using consensus-free (DynaStore, SmartMerge, SpSn and FreeStore with $\mathcal{L}$) and consensus-based (RAMBO and FreeStore with $\mathcal{P}$) algorithms. We present the number of communication steps for the best case scenario – when all processes propose the same updates in a reconfiguration – and for the worst case – when each process proposes different updates. In order to simplify our analysis, we consider that no reconfiguration is started concurrently with the one we are analyzing. Similarly, we assume a synchronous execution of the Paxos protocol [19], which requires only three communication steps, for consensus-based algorithms.

FreeStore reconfiguration is significantly more efficient than previous consensus-based and consensus-free protocols. FreeStore with $\mathcal{P}$ requires two less communication steps than RAMBO and FreeStore with $\mathcal{L}$ outperforms other consensus-free approaches by almost an order of magnitude. In particular, FreeStore with $\mathcal{L}$ presents the best performance among all considered reconfiguration protocols in the best case, which is expected to be the norm in practice. An open question is if this number constitutes a lower bound.

7 Experimental Evaluation

In this section we present an experimental evaluation of FreeStore (1) to quantify its overhead when compared with the (static) ABD protocol in periods without reconfigurations, and (2) to assess the negative impact of a reconfiguration in the system performance.

We implemented prototypes of the ABD [4], FreeStore and DynaStore [2] protocols in the *Go* programming language and conducted two sets of experiments in Emulab [22]: a micro-benchmark designed to evaluate the read and write throughput and latency in absence of reconfiguration; and an execution showing the performance of dynamic algorithms during faults and reconfigurations. We chose DynaStore to represent existing consensus-free dynamic protocols [2, 12, 17] to show that design decisions such as *checking a set of SWMR static registers to verify if some reconfiguration occurred before executing each r/w* and *coupling the execution of r/w and reconfigurations* have a significant impact in the system performance. Although comparing with DynaStore suffice for these goals, the interested reader can find a more extensive experimental evaluation of reconfiguration protocols in [16].

Experimental Setup. We used 18 *pc3000* (3.0 GHz 64-bit Pentium Xeon, 2GB of RAM and gigabit network cards) connected by a 100Mb switched network. We run each server in a separated machine, while the clients were uniformly distributed among the remaining machines. The software installed was Fedora 15 64-bit with *kernel* 2.6.20 and *Go* 1.2.

Micro-benchmarks. We start by reporting the latency and throughput results for the system configured with 3 servers (Figure 2). We used up to 18 clients to read or write a value of 512 bytes. Both latency and throughput was measured at the clients: the latency is the mean time demanded to perform a r/w operation discarding the 5% values with greater variance; the throughput is the sum of all client operations completed in a interval.

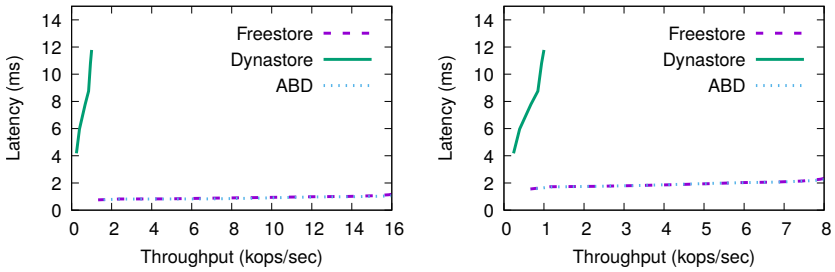


Figure 2 Latency vs. throughput for read (left) and write (right) operations; $n = 3$ and $f = 1$.

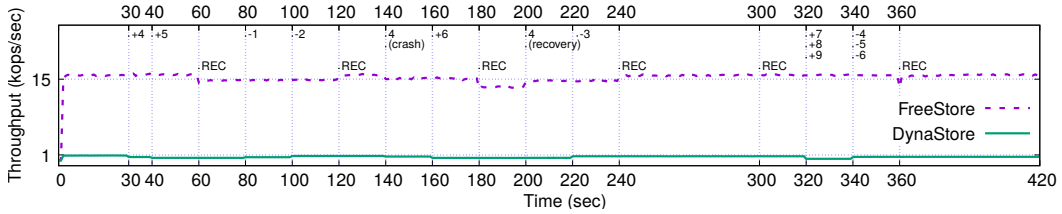


Figure 3 Throughput evolution across faults and reconfigurations for FreeStore and DynaStore.

These experiments show that FreeStore imposes a negligible overhead to the static ABD protocols. This happens because these protocols are very similar: the difference is that FreeStore messages must carry the current view to check if a client is using an updated view (we used hashes of the views; the complete views were used only when necessary an update in the client view). In contrast, DynaStore performs poorly because it must access a set of static SWMR registers to check for view updates before executing each r/w operation.

Reconfigurations and faults. This experiment considers the behavior of FreeStore with $\mathcal{L}$ (the result is similar with $\mathcal{P}$) and DynaStore protocols under reconfigurations, failures and recoveries. We observed how the throughput of these systems evolve over several events when a demanding workload is applied. We used an initial view with 3 servers ($v_0 = \{1, 2, 3\}$) and 18 clients that keep reading a value of 512 bytes over the course of 420 seconds. The results in Figure 3 show that FreeStore significantly outperforms DynaStore.

In the experiment, servers 4-9 asked to join at times 30, 40, 160, and 320 (servers 7-9), respectively, while servers 1-6 asked to leave at times 80, 100, 220, 340 (servers 4-6), respectively. DynaStore reconfigurations occurred each time an update was requested [2], while FreeStore reconfigurations were configured to occur periodically at each 60 seconds (notice that at time 360, a FreeStore reconfiguration replaces all servers in the system). Moreover, server 4 crashed and recovered at times 140 and 200, respectively.

This experiment shed light in how reconfigurations impact the performance of concurrent r/w operations. The mean time required for a FreeStore reconfiguration was 19 ms, with r/w operations blocked for only 4 ms (see Algorithm 3). Increasing the size of the value stored in the system may lead this time to increase. However, in all previous works on asynchronous reconfigurations [2, 12, 17], the state transfer happens during a r/w operation and, consequently, the time to finish these operations will also increase.

8 Conclusions

This paper presented a new approach to reconfigure fault-tolerant storage systems, which clarifies the differences between relying or not on consensus for agreement in the next view to be installed. The main result is a protocol that is simpler and cheaper (in terms of communication steps for either r/w operations or reconfigurations) than the previously proposed solutions. Furthermore, our approach fully decouples the execution of r/w operations and reconfigurations, imposing a negligible overhead to the static ABD protocol. Another interesting result is that, in the best case, FreeStore consensus-free reconfiguration is faster than protocols based on consensus.

A final contribution of this work is the introduction of a new abstraction called view generator. We believe that exploring different instantiations of this abstraction and their properties is an important avenue for future work.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of EATCS: The Distributed Computing Column*, 2010.

2 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58:7:1–7:32, 2011.

3 Eduardo Alchieri, Alysson Bessani, Fabiola Greve, and Joni Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. *ArXiv*, 2016. `arXiv:1607.05344`.

4 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

5 Kenneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles (SOSP'87)*, 1987.

6 C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd Edition)*. Springer-Verlag, 2011.

7 Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

8 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Fast access to distributed atomic memory. *SIAM Journal on Computing*, 39(8):3752–3783, 2010.

9 Cyntia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322, 1988.

10 Rui Fan and Nancy Lynch. Efficient replication of large data objects. In *Proc. of the 17th Int. Symp. on Distributed Computing (DISC'03)*, 2003.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

12 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proc. of the 29th Int. Symp. on Distributed Computing (DISC'15)*, 2015.

13 David Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symp. on Operating Systems Principles (SOSP'79)*, 1979.

14 Seth Gilbert, Nancy Lynch, and Alex Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4), 2010.

15 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1), 1991.

16 Leander Jehl and Hein Meling. The case for reconfiguration without consensus. In *Proc. of the 20th Int. Conf. on Principles of Distributed Systems (OPODIS'16)*, 2016.

17 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *Proc. of the 29th Int. Symp. on Distributed Computing (DISC'15)*, 2015.

18 Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, 1986.

19 Leslie Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, 1998.

20 Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic Byzantine storage. In *Proc. of the 34th Int. Conf. on Dependable Systems and Networks (DSN'04)*, 2004.

21 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic Reconfiguration: A Tutorial. In *Proc. of the 19th Int. Conf. on Principles of Distributed Systems (OPODIS'15)*, 2015.

22 Brian White et. al. An integrated experimental environment for distributed systems and networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementations (OSDI'02)*, 2002.

Hardening Cassandra Against Byzantine Failures*

Roy Friedman[1] and Roni Licher[2]

1 **Department of Computer Science, Technion, Haifa, Israel**
`roy@cs.technion.ac.il`
2 **Department of Computer Science, Technion, Haifa, Israel**
`ronili@cs.technion.ac.il`

Abstract

Cassandra is one of the most widely used distributed data stores. In this work, we analyze Cassandra's vulnerabilities when facing Byzantine failures and propose protocols for hardening Cassandra against them. We examine several alternative design choices and compare between them both qualitatively and empirically by using the Yahoo! Cloud Serving Benchmark (YCSB) performance benchmark.

Some of our proposals include novel combinations of quorum access protocols with MAC signatures arrays and elliptic curve public key cryptography so that in the normal data path, there are no public key verifications and only a single relatively cheap elliptic curve signature made by the client. Yet, these enable data recovery and authentication despite Byzantine failures and across membership configuration changes. In the experiments, we demonstrate that our best design alternative obtains roughly half the performance of plain (non-Byzantine) Cassandra.

1998 ACM Subject Classification C.2.4 Distributed Systems, K.6.5 Security and Protection, B.8.1 Reliability, Testing, and Fault-Tolerance

Keywords and phrases Cassandra, Byzantine Fault Tolerance, Distributed Storage

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.27

1 Introduction

Distributed data stores are commonly used in data centers and cloud hosted applications, as they provide fast, reliable, and scalable access to persistently stored data. Persistent storage is fundamental in most applications, yet developing an effective one is notoriously difficult.

Due to inherent tradeoffs between semantics and performance [8] as well as the desire to offer various flexible data management models, a plethora of products has been developed. These differ in the data access model, which can range from traditional relational databases, to wide-columns [12, 28], key-value stores [1, 17], as well as graph databases and more. Another axis by which such systems differ is the consistency guarantees, which can range from strong consistency [29] to eventual consistency [36] and a myriad of intermediate options.

In our work, we focus on Cassandra [28]. Cassandra follows the wide-column model, and offers very flexible consistency guarantees. Among open source data stores, it is probably the most widely used; according to the Cassandra Apache project page [5], more than 1,500 companies are currently using Cassandra, including, e.g., Apple, CERN, Comcast, eBay, GitHub, GoDaddy, Hulu, Instagram, Intuit, Microsoft, Netflix, Reddit, and more.

Cassandra can effectively withstand benign failures, but it was not designed to overcome Byzantine attacks, in which some nodes in the system may act arbitrarily, including in

* A full version of the paper is available at [22], `https://arxiv.org/abs/1610.02885`.

a malicious manner. Handling Byzantine failures requires sophisticated protocols and more resources. However, ever since the seminal PBFT work of Castro and Liskov [11], the practicality of building Byzantine fault tolerant replicated state machines has been demonstrated by multiple academic projects, e.g., [13, 24] to name a few. Interestingly, storage systems offer weaker semantics than general replicated state machines, and therefore it may be possible to make them resilient to Byzantine failures using weaker timing and failure detection assumptions, as proposed in [10, 31, 33]. Yet, to the best of our knowledge, we are the first to systematically harden Cassandra against Byzantine failures.

Contributions

We analyze Cassandra's structure and protocols to uncover their vulnerability to Byzantine behavior. We then propose alterations to Cassandra's existing protocols that overcome these failures. We examine several alternative solutions and compare between them qualitatively and quantitatively. Let us emphasize that one of our main design principles is to preserve Cassandra's basic interaction model as much as possible, to increase the likelihood of adoption and to minimize the number of lines of code we need to change. After all, our goal is to harden the existing system, not creating a new one.

We have benchmarked the original Cassandra and our hardened versions of Cassandra using the standard YCSB benchmark [14]. As expected, a key factor to obtain reasonable performance is in the type of cryptography used. E.g., using traditional RSA signatures dramatically lowers the performance. Conversely, using only (symmetric key) MAC signatures is challenging when data must survive configuration changes and be shared by multiple clients. To that end, here we propose a novel combination of vectors of MACs with the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [27]. The normal data path of this proposal incurs no public key verifications and only a single relatively cheap ECDSA signature made by the client. Yet, these enable data recovery and authentication despite Byzantine failures and across membership configuration changes. The measured performance of the best configuration of our hardened Cassandra, utilizing our new approach, is only twice worse than the original Cassandra; an order of magnitude better than the naive approach.

Note that while using an array of MAC signatures instead of PKI has been done in the past, e.g., in [11], prior use of this idea was restricted to ordering decisions that are contained within a single configuration. In contrast, in our work we ensure data survivability across configuration changes, which requires the extra protection of the elliptic curve signature.

2 Related Work

The seminal work of Castro & Liskov [11] inspired multiple extensions. Clement et al. [13] introduced UpRight, a modular library for BFT replicated state machine. BFT-SMaRt [7] and Prime [3] have improved the performance when facing Byzantine behaviour, whereas Abstract [24] is the state of the art in BFT replicated state machine. It adds the ability to abort a client request when faults occur and then dynamically switch to another BFT protocol that produces better results under the new system conditions. The work of [35] explored adding BFT resilience to transactional systems that follow the scalable differed update replication model.

Malkhi & Reiter [33] were the first to discuss *Byzantine* quorum systems, i.e., using read and write quorums such that any two quorums intersect in at least one *correct* node. Furthermore, the system remains available in spite of having up to f Byzantine nodes.

Aguilera & Swaminathan [2] explored BFT storage for slow client-server links while relying on synchronized clocks. They designed a *linearizable abortable* register in which partial writes due to benign client failures do not have any effect by using unique timestamps and timestamp promotion when conflicts appear. Yet, they did not show an actual implementation nor performance analysis. As our work preserves Cassandra's semantics, we are able to design faster operations requiring lighter cryptography measures even when conflicts occur.

Byzantine clients might try to perform *split-brain-writes*, i.e., writing multiple values to different servers using the same timestamp. Preventing this can be done by obtaining a commitment from a quorum to bind a timestamp and a value on every write. In Malkhi & Reiter's approach [33], on every write, the servers exchange inter-servers messages agreeing on the binding. In Liskov & Rodrigues's approach [31], the servers transmit signed agreements to the client that are later presented to the servers as a proof for the quorum agreement. In contrast, we do not prevent split-brain-writes, but rather repair the object's state on a read request (or in the background).

Some BFT cloud storage systems provide *eventual consistency* [36]. Zeno [39] ensures *causal order consistency* [29] with at least $f + 1$ correct nodes. Depot [32] tolerates any number of Byzantine clients and servers and guarantees *Fork-Join-Causal order consistency*.

Aniello et al. [4] explored Byzantine DoS attacks on gossip based membership protocols. They have demonstrated their attack on Cassandra [28] and presented a way to prevent it by using signatures on the gossiped data. Other more general solutions for BFT gossip membership algorithms were shown in Fireflies [26] and Brahms [9]. The first uses digital signatures, full membership view and a pseudorandom mesh structure and the latter avoids digital signatures by sophisticated sampling methods.

Sit & Morris [40] mapped classic attacks on *Distributed Hash Tables* (DHT). Some of the attacks can be disrupted by using SSL. Other attacks described in [40], such as storage and retrieval attacks, are addressed in our work.

Non malicious value and state error failures in Cassandra and other distributed storage systems have been explored in [15, 23]. Solutions to such problems can be more efficient than Byzantine resilient implementations, but inherently only protect against a more restrict failure scenario.

3 Model and Assumptions

We assume a Cassandra system consisting of *nodes* and *clients*, where each may be *correct* or *faulty* according to the Byzantine failure model [30]. A correct entity acts according to its specification while a faulty entity can act arbitrarily, including colluding with others.

In our proposed solutions, we assume that the maximal number of faulty nodes is bounded by f. We initially assume that all clients are *correct*, but later relax this assumption. When handling Byzantine clients, we do not limit the number of faulty clients nor change the assumption on the maximal number of f faulty nodes. Yet, we assume that clients can be authenticated so correct nodes only respond to clients that are allowed to access the system according to some verifiable *access control list* (ACL). We use the terms nodes and processes interchangeably and only to refer to Cassandra nodes.

We assume a fully connected partially synchronous distributed system. Every node can directly deliver messages to every other node and every client can directly contact any system node. We also assume that each message sent from one correct entity to another will eventually arrive exactly once and without errors. That can be implemented, e.g., on top of fair lossy networks, using retransmission and error detection codes. We do not assume any

bound on message delay or computation time in order to support our safety and liveness properties. However, efficiency depends on the fact that most of the time messages and computation steps do terminate within bounded time [19].

Every system entity has a verifiable PKI certificate. We assume a trusted *system administrator* that can send signed membership configuration messages.

The system shares a loosely synchronized clock which enables detection of expired PKI certificates in a reasonable time but is not accurate enough to ensure coordinated actions. We discuss this clock in Appendix B.3.

4 Brief Overview of Cassandra

Cassandra stores data in tables with varying number of columns. Each node is responsible for storing a range of rows for each table. Values are replicated on multiple nodes according to the configurable *replication factor*.

Mapping data to nodes follows the *consistent hashing* principle, where nodes are logically placed on a virtual ring by hashing their ids. In fact, each node is represented as multiple *virtual nodes* [17]. Each virtual node generates a randomized key on the ring, called a *token*, representing its *place*. A virtual node is responsible for hashed keys that fall in the range from its place up to the next node on the ring, known as its *successor*. Each node also stores keys in the ranges of the $N-1$ preceding nodes, where N is the replication factor parameter. The N nodes that should store a given value are called its *replication set*.

Cassandra uses a *full membership view*, where every node knows every other node. A node that responds to communication is considered *responsive* and otherwise it is *suspected*. Nodes exchange their views via *gossip* [41]. The gossip is disseminated periodically and randomly; every second, each node tries to exchange views with up to three other nodes: one responsive, one suspected, and a *seed* [28]. A new node starts by contacting seed nodes.

Cassandra provides tunable consistency per operation. On every operation, the client can specify the *consistency level* that determines the number of replicas that have to acknowledge the operation. Some of the supported consistency levels are: *one* replica, a *quorum* [25] of replicas and *all* of the replicas. According to the consistency level requested in the writes and in the respective reads, *eventual consistency* [36] or *strong consistency* can be achieved.

On each operation, a client connects to any node in the system. This selected node acts as a *proxy* on behalf of the client and contacts the relevant nodes using its view of the system. In the common configuration, the client selects a proxy among all system nodes in a *Round Robin* manner. The proxy node may contact up to N nodes that are responsible for storing the value according to the requested consistency level. If the required threshold of responses is satisfied, the proxy will acknowledge the write or forward the latest value, according to the stored timestamp, to the client. If the proxy fails to contact a node on a write, it stores the value locally and tries to update the suspected node at a later time. The stored value is called *hinted handoff* [16]. If a proxy receives multiple versions on a read query, it performs a *read repair* to update nodes that hold a stale version with the most updated one. As of Cassandra 1.0, this replaces the earlier *sloppy quorums* [17] approach.

If a node is unresponsive for a long time, hinted handoffs that were saved for this node may be deleted. Similarly, a hinted handoff may not reach its targeted node if the node that stores it fails. This is overcome with Cassandra's manual *anti-entropy* tool, where nodes compute and exchange *Merkle trees* for their values and sync the outdated ones.

The primary language for communicating with Cassandra is the *Cassandra Query Language* (CQL) [16]. Here, we focus on put and get commands as available in standard NoSQL key-value databases and ignore other options.

5 Hardened Cassandra

We identify Byzantine vulnerabilities in Cassandra and suggest ways to overcome them.

5.1 Impersonating

Cassandra supports SSL. Yet, sometimes messages need to be authenticated by a third party, e.g., a read response sent from a node to a client through a proxy node, which we support through digital signatures. In both SSL and digital signatures, we depend on PKI.

Digital signatures are divided into two main categories: *public/private keys* vs *MAC tags*. Public key signatures are more powerful as they enable anyone to verify messages. The downside of public key signatures is their compute time, which is about 2-3 orders of magnitude slower than MAC tags and these signatures are significantly larger, e.g., RSA 2048b versus AES-CBC MAC 128b.

5.2 Consistency Level

Recall that Cassandra offers a configurable replication factor N as well as the number of nodes that must acknowledge each read (R) and each write (W). This threshold can be one node or a quorum (majority in Cassandra) or all N nodes. When up to f nodes are Byzantine, querying fewer than $f+1$ nodes may retrieve old data (signed data cannot be forged), violating the consistency property. On the other hand, querying more than $N-f$ nodes may result in loss of availability. We present two approaches: (1) using Byzantine quorums for obtaining Strong Consistency and (2) using Cassandra quorums with a scheduled run of the anti-entropy tool for obtaining *Byzantine Eventual Consistency*.

Byzantine Quorums

By requesting that each read and each write will intersect in at least $f+1$ nodes, we ensure that every read will intersect with every write in at least one *correct* node. That is, $R+W \geq N+f+1$. As for liveness, we must require that $R \leq N-f, W \leq N-f$. By combining these requirements, we obtain: $N \geq 3f+1$.

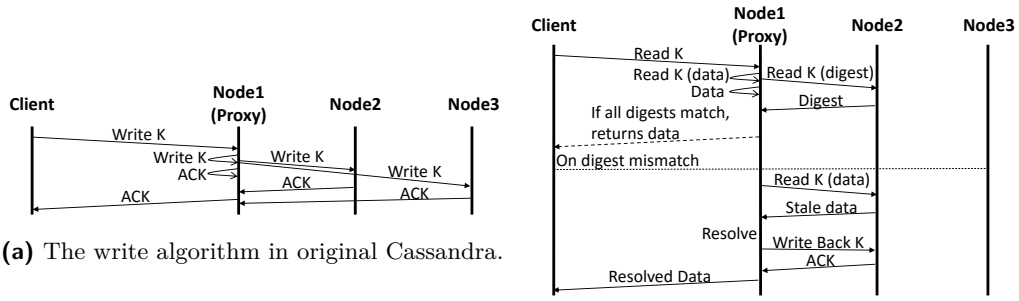
The last bound was formally proved by Malkhi & Reiter [33]. Cachin et al. [10] have lowered this bound to $2f+1$ by separating between the actual data and its *metadata*; storing the medadata still requires $3f+1$ nodes. The above separation was presented under the assumptions of benign writers and Byzantine readers.

Cachin's solution is beneficial for storing large data items. Yet, when storing small values, the method of [10] only increases the overhead. A system may offer either solution according to its usage, or employ both in a hybrid way, according to each value's size.

Byzantine Eventual Consistency

For eventual consistency, all replication set nodes must eventually receive every update. Further, writes order conflicts should be resolved deterministically. Here, there is no bound on the propagation time of a write, but it should be finite. In particular, if no additional writes are made to a row, eventually all reads to that row will return the same value.

Byzantine eventual consistency can be obtained through majority quorums. In this approach, the replication set is of size $2f+1$ nodes while write and read quorums are of size of $f+1$. Hence, each write acknowledged by $f+1$ nodes is necessarily executed by at least one correct node. This node is trusted to update the rest of the nodes in the background.



(a) The write algorithm in original Cassandra.

(b) The read algorithm in original Cassandra.

Figure 1 Original Cassandra. Configuration: N=3 and R=2.

As this node is correct, it will eventually use the anti-entropy tool to update the rest of the replication set. Recall that the client request is signed so the servers will be able to authenticate this write when requested.

Every read is sent to $f + 1$ nodes and thus reaches at least one correct node. This correct node follows the protocol and accepts writes from proxy nodes and from the anti-entropy tool. So, eventually, it retrieves the latest update. Due to the cryptographic assumptions, a Byzantine node can only send old data and cannot forge messages. Hence, on receiving a value from the anti-entropy tool that does not pass the signature validation, we can use it as a Byzantine failure detector and notify the system administrator.

5.3 Proxy Node

Figures 1a and 1b present the current write and read flows in Cassandra, including the role of proxies. A Byzantine proxy node can act in multiple ways, such as (1) respond that it has successfully stored the value without doing so, (2) perform a split-brain-write, and (3) respond that the nodes are not available while they are. We augment the existing flows of writing and reading in Cassandra to overcome these vulnerabilities below.

Naive Hardened Write Operation

Our modified write algorithm appears in Figures 2 and 8. Here, when writing a new value, the client signs the value and a node will store it only if it is signed by a known client according to the ACL with a timestamp that is considered fresh (configurable). On each store, the storing node signs an acknowledgment that the client can verify. The signed acknowledgment also covers the timestamp provided by the client, preventing replay attacks by the proxy. A client completes a write only after obtaining the required threshold of signed responses, which now the proxy cannot forge. If one proxy fails to respond with enough signed acknowledgments in a configurable reasonable time, the client contacts another node to serve as an alternative proxy for the operation. After contacting at most $f + 1$ proxy nodes (when needed), the client knows that at least one *correct* proxy node was contacted.

Naive Hardened Read Operation in Details

The read algorithm has three parts: (1) Reading data from one node and only a digest from the rest. As an optimization, the read may target only a known live quorum instead of all relevant nodes. (2) On digests mismatch, a full read is sent to all contacted nodes from the

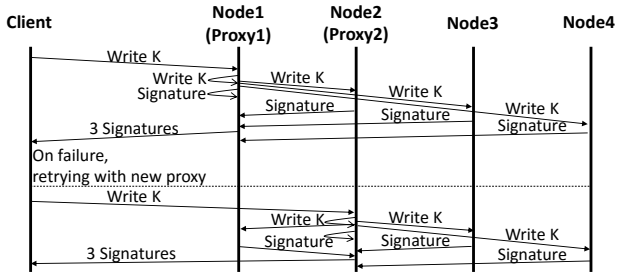


Figure 2 Illustrating our write algorithm from Figure 8 where the proxy verifies each store acknowledgment. Configuration: N=4 and W=3.

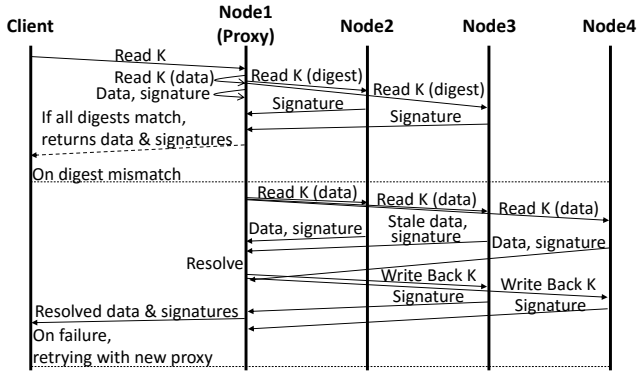


Figure 3 Illustrating our read algorithm from Figure 9 where the proxy verifies. N=4 and R=3.

first phase, retrieving the data. (3) The proxy resolves the conflict by creating a row with the most updated columns according to their timestamps, using lexicographical order of values as tie breakers when needed. The resolved row is sent back to out-dated nodes.

Figures 3 and 9 present our modified read algorithm, including the following changes: (1) In case the first phase is optimized by addressing only a known live quorum of nodes, if a failure occurs, we do not fail the operation but move to a full read from all nodes. Thus, if a Byzantine node does not respond correctly, it does not fail the operation. (2) If there is a digest mismatch in the first phase, we do not limit the full read only to the contacted nodes from the first phase but rather address all replication set nodes. Hence, Byzantine nodes cannot answer in the first phase and fail the operation by being silent in the second phase. (3) During resolving, the nodes issue a special signature, notifying the client about the write back. The proxy then supplies the original answers from the first phase to the client, all signed by the nodes. This way, the client can authenticate the resolving's correctness.

The motivation for forwarding the original answers from the proxy to the client relates to the fast write optimization of Cassandra, where all writes are appended to a commit log and reconciled in the background or during a following read request. Hence, while stale values are saved, if there is already a newer value, a stale value would not be served by any read. Unfortunately, this optimization exposes an attack by which a Byzantine proxy would request a quorum of nodes to store an old value. By providing the client with the original answers, it can verify that the write-back was necessary and correct.

5.3.1 Targeting Irrelevant Nodes

Byzantine proxies may direct read requests to irrelevant nodes, who will return a verifiable empty answer. We consider three remedies for this: (1) Using clients that have full membership view, already supported by Cassandra, so a client knows which nodes have to respond. (2) Using an authentication service that is familiar with the membership view and can validate each response. A client can use this service to authenticate answers. (3) Configure the nodes to return a signed 'irrelevant' message when requested a value that they are not responsible for. Here, clients avoid counting such 'irrelevant' answers as valid responses.

A Byzantine proxy can forward updates only to members of a single write quorum to decrease long term availability. The anti-entropy tool is periodically invoked to overcome this. It requires each value to be accompanied by a correct client signature for authentication.

5.3.2 Proxy Acknowledgments Verification

In our proposed solution as presented so far, we have requested the proxy to verify the nodes acknowledgments and accept a response only if it is signed correctly. Yet, both hardened read and write protocols forward to the client digitally signed acknowledgments so the latter can authenticate the completion of the operation by the nodes. This motivates shifting the entire verification from the proxy to the client, to reduce the proxy load. Below, we identify the challenges in enabling this optimization and offer solutions:

1. Consider a correct proxy and f Byzantine nodes. The Byzantine nodes may reply faster with bad signatures (they need not verify signatures nor sign). The proxy then returns to the client $f + 1$ good signatures and f bad signatures. Contacting an alternative proxy now might produce the same behavior.
2. Consider a colluding Byzantine proxy that is also responsible to store data itself. On a write, the proxy asks the Byzantine nodes to produce a correct signature without storing the value. The proxy also asks one correct node to store the data and produces false f signatures for some nodes. The client will get $f + 1$ correct signatures and f bad signatures, while only one node really stored the value.

To overcome the above, we let the client contact the proxy again in case it is not satisfied with the $2f + 1$ responses it obtained. On a read, the client requests the proxy to read again without contacting the nodes that supplied false signatures. On a write, the client requests the proxy to fetch acknowledgments from additional nodes.

As mentioned above, the motivation for this alternative is that signatures verification is a heavy operation. In the proxy verification option, on every write, the proxy is required to perform at least $2f + 1$ signature verifications. In the alternative client only verification option, the latency penalty will be noticed only when Byzantine failures are present and could be roughly bounded by the time of additional RTT (round-trip-time) to the system and f parallel RTT's inside the system (counted as one), multiplying it all by f (the number of retries with alternative proxies). Assuming that in most systems Byzantine failures are rare, expediting the common correct case is a reasonable choice.

The client only verification option also enables using MAC tags instead of public signatures, since only the client verifies signatures. To that end, a symmetric key for each pair of system node and client should be generated. Every client has to store a number of symmetric keys that is equal to the number of system nodes. Every node has to store a number of symmetric keys that is equal to the number of (recently active) clients. These keys can be pre-configured by the system administrator or be obtained on the first interaction through a secure SSL line. This yields significant speedups both for the node signing and for the client verification. The exact details appear in [22].

5.3.3 Proxy Resolving vs. Client Resolving

Recall that when Cassandra's read operation detects an inconsistent state, a resolving process is initiated to update outdated replicas. This way, the chance for inconsistency in future reads decreases. In plain Cassandra as well as in our solution as presented so far, the proxy is in charge of resolving such inconsistent states. An alternative option is to let the client resolve the answers and write back the resolved value using a write request that specifies to the proxy which replicas are already updated.

To prevent Byzantine nodes from manipulating the resolver with false values, the resolver must verify the client signature on each version. When combining the client resolving option with using a proxy that is not verifying nodes acknowledgments (as discussed in Section 5.3.2), the proxy no longer needs to verify any client signatures, improving its scalability. The exact details appear in [22].

5.3.4 From Public Key Signatures to MAC Tags

The use of public key signatures has a major performance impact while switching to MAC tag is not trivial. In Section 5.3.2, we described how to switch from public key signatures to MAC tags on messages sent from nodes to clients.

Supporting MAC tags on messages sent from clients to nodes poses the following challenges: (1) Joining new nodes to Cassandra. These nodes have to fetch stored values for load-balancing. As a client does not know who these future nodes are, it cannot prepare MAC tags for them. One solution is that a new node will only store values that were found in at least $f+1$ nodes. Alternatively, the client may re-write all values that the new node has to store. (2) Using the anti-entropy tool and resolving consistency conflicts need to ensure the correctness of the values by contacting at least $f+1$ nodes that agree on the values. Alternatively, every node will have to store a vector of MAC tags for each responsible node. Storing a signature vector poses another challenge: a Byzantine proxy can manipulate the signatures vector sent to each node, leaving only the node's signature correct and corrupting all other nodes' signatures (making the stored vector useless). This can be overcome by adding another MAC tag on the signatures vector, proving to the node that the tags vector was not modified by the proxy.

Due to these issues and in order to speed up the write path, we suggest a hybrid solution as presented in Figure 4. A write is signed with a public key signature, which is covered by MAC tags, one for each node. A node then verifies only the MAC tag and stores only the public key signature. Hence, in the common case, we use a single public key signature with no public key verifications at all. When things go bad, we fall back to the public key signature. Furthermore, use of ECDSA [27] offers much faster signing than *RSA* [37].

Finally, when using MAC tags on the client to node path, the client should know what are the relevant nodes for that key. One solution is to ensure clients are updated about the nodes tokens. This way, on every write, the client knows what keys to use. Since our solution has a fall back option, even if there is a topology change that the client is late to observe, the new node (targeted by the proxy) can still use the public signature and not fail the write. On the write acknowledgment, the new node can update the client about the (signed) topological change.

5.3.5 Comparing The Variants

Tables 1, 2 and 3 summarize the alternatives proposed in this section. We focus on the number of signing and verification operations of digital signatures as these are the most

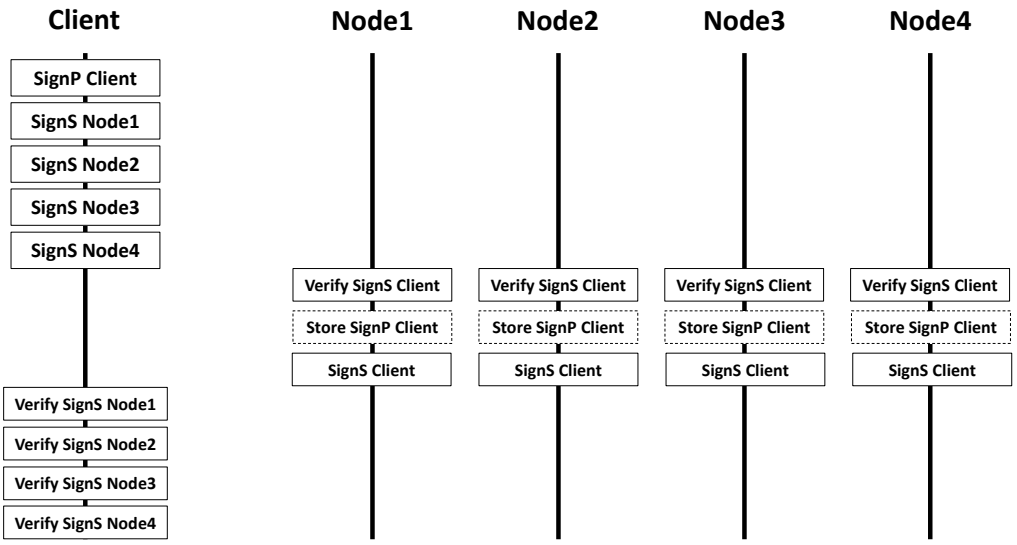


Figure 4 Illustration of our hybrid signing solution. The *SignP* stands for public key signature, using the private key of the signing entity. The *SignS* stands for MAC tag.

Table 1 Comparing the variants of our solution with the most optimist assumptions. C is the number of columns, (p) indicated public key signatures and (s) MAC tags. When the proxy does not verify, we refer both to the proxy resolves and client resolves modes. We assume that on a read, the proxy uses the optimization in the first phase and contacts only a Byzantine quorum and not all replicas. For example, the forth row presents a proxy that does not verify acknowledgments and MAC tags are used from client to nodes and from nodes to client. In this variant, the client signs the C columns using public key signatures and adds $3f + 1$ MAC tag, one for each node. All nodes $(3f + 1)$ have to store it and they verify only their MAC tags. All nodes issue verifiable acknowledgments $(3f + 1)$ and the client verifies only a Byzantine quorum $(2f + 1)$.

Proxy Verifies?	Op	MAC Tags	Signatures	Verifications
Yes	Write	None	Client: C(p) Nodes: $3f + 1$(p)	Nodes: $(3f + 1) \cdot C$(p) Proxy: $2f + 1$(p) Client: $2f + 1$(p)
No	Write	None	Client: C(p) Nodes: $3f + 1$(p)	Nodes: $(3f + 1) \cdot C$(p) Client: $2f + 1$(p)
No	Write	Nodes to client	Client: C(p) Nodes: $3f + 1$(s)	Nodes: $(3f + 1) \cdot C$(p) Client: $2f + 1$(s)
No	Write	Both ways	Client: C(p) & $3f + 1$(s) Nodes: $3f + 1$(s)	Nodes: $3f + 1$(s) Client: $2f + 1$(s)
Yes	Read	None	Nodes: $2f + 1$(p)	Proxy: $2f + 1$(p) Client: $2f + 1$(p)
No	Read	None	Nodes: $2f + 1$(p)	Client: $2f + 1$(p)
No	Read	Nodes to client	Nodes: $2f + 1$(s)	Client: $2f + 1$(s)

time consuming. We divide our analysis in three: (1) best case and no failures, (2) a benign mismatch on a read that requires resolving, and (3) worst case with f Byzantine nodes.

5.4 Handling Byzantine Clients

Notice that some Byzantine clients actions are indistinguishable from correct clients behaviors. For example, erasing data or repeatedly overwriting the same value. Yet, this requires the client to have ACL permissions.

Here, we focus on preserving data consistency for correct clients. I.e., a correct client should not observe inconsistent values resulting from a split-brain-write nor should it read older values than values returned by previous reads.

Specifically, we guarantee the following semantics, as in plain Cassandra: (1) The order between two values with the same timestamp is their lexicographical order (breaking ties

Table 2 Comparing the variants in the read flow in case of a benign mismatch that requires resolving. C is the number of columns, M is the number of outdated replicas in the used quorum, (p) indicated public key signatures and (s) MAC tags. We assume that the proxy uses the optimization in the first phase and contacts only a Byzantine quorum. For example, the first row presents a proxy that verifies the acknowledgments and resolves conflicts when mismatch values are observed. MAC tags are not in use. On a read request, a Byzantine quorum of nodes $(2f+1)$ have to retrieve the row and sign it. The proxy verifies their signatures $(2f+1)$ and detects a conflict. Then, the proxy requests all relevant nodes (except for the one that returned data in the first phase) for the full data ($3f$ nodes sign and the proxy verifies only $2f$). The proxy resolves the mismatch (verifies C columns) and sends the resolved row to the M outdated nodes (write-back). These nodes verify the row (C) and sign the acknowledgments that are later verified by the proxy. The proxy supply the client with the original $2f+1$ answers and the resolved row signed also by M nodes that approved the write-back.

Proxy Verifies?	Mismatch Resolving	MAC tags	Signatures	Verifications
Yes	Proxy	No	Nodes: $5f+1+M$(p)	Nodes: $M \cdot C$(p) Proxy: $4f+1+C+M$(p) Client: $2f+1+M$(p)
No	Proxy	No	Nodes: $5f+1+M$(p)	Nodes: $M \cdot C$(p) Proxy: C(p) Client: $2f+1+M$(p)
No	Proxy	Yes	Nodes: $5f+1+M$(s)	Nodes: $M \cdot C$(p) Proxy: C(p) Client: $2f+1+M$(s)
No	Client	No	Nodes: $5f+1+M$(p)	Nodes: $M \cdot C$(p) Client: $2f+1+C+M$(p)
No	Client	Yes	Nodes: $5f+1+M$(s)	Nodes: $M \cdot C$(p) Client: $2f+1+M$(s) & C(p)

Table 3 Comparing the variants of the read and write flows in the worst case f Byzantine nodes. Due to the wide options of Byzantine attacks and the fact that every Byzantine node can waste other node's cycles, we compare the variants only from the point of view of a correct client. C is the number of columns, M is the number of outdated replicas in the used quorum, (p) indicated public key signatures and (s) MAC tags. For example, the second row presents a proxy that does not verify the acknowledgments in a write operation. MAC tags are not in use. On a write request, the client signs the C columns and sends it to the proxy. The client receives from the proxy responses from a Byzantine quorum of nodes $(2f+1)$ and detects that one is incorrect. The client requests the proxy f more times for the missing signature and every time gets a false signature. Then, the client uses alternative proxies repeatedly f additional times. At last, the client successfully retrieves all $2f+1$ correct signatures due to our assumption on f.

Proxy Verifies?	Op	Mismatch Resolving	MAC Tags	Signatures	Verifications	Client-Proxy Requests
Yes	Write	-	None	C(p)	$(2f+1)(f+1)$(p)	$f+1$
No	Write	-	None	C(p)	$(3f+1)(f+1)$(p)	$(f+1)(f+1)$
No	Write	-	Nodes to client	C(p)	$(3f+1)(f+1)$(s)	$(f+1)(f+1)$
No	Write	-	Both ways	C(p)	$(3f+1)(f+1)$(s)	$(f+1)(f+1)$
Yes	Read	Proxy	None	None	$(2f+1+M)(f+1)$(p)	$(f+1)$
No	Read	Proxy	None	None	$(2f+1+M)(f+1)(f+1)$(p)	$(f+1)(f+1)$
No	Read	Client	None	None	$(2f+1)(f+1)(f+1)+C+(M+f)(f+1)$(p)	$(f+1)(f+1)+(M+f)(f+1)$
No	Read	Proxy	Nodes to client	None	$(2f+1+M)(f+1)(f+1)$(s)	$(f+1)(f+1)$
No	Read	Client	Nodes to client	None	$(2f+1)(f+1)(f+1)+(M+f)(f+1)$(s) & C(p)	$(f+1)(f+1)+(M+f)(f+1)$

according to their value). (2) A write of multiple values with the same timestamps is logically treated as multiple separate writes with the same timestamp. (3) Deleting values is equivalent to overwriting these values with a *tombstone*. (4) A read performed by a correct client must return any value that is not older (in timestamp order) than values returned by prior reads. (5) A read performed after a correct write must return a value that is not older (in timestamp order) than that value.

As mentioned, if the proxy handling a read observes multiple versions from different nodes, it resolves the mismatch and writes the resolved value back to the nodes. The resolved version will be a row with the most updated columns according to their timestamps. If the proxy observes two values with the same timestamp, it will use the lexicographical order of the values as a tie breaker.

For split-brain-writes, Byzantine clients colluding with Byzantine proxies may sign

multiple values with the same timestamp. Proxies can send these different values with the same timestamps to different nodes for a split-brain-write. Notice that a split-brain-write could occur spontaneously in plain Cassandra by two correct clients that write in parallel since in Cassandra clients provide the write's timestamp, typically by reading their local clock.

Consider a Byzantine client colluding with a proxy to perform a split-brain-write. Due to the resolve mechanism, a read that sees both values returns only the latest value in lexicographical order. Further, on a client read, the proxy resolves the conflict and updates a quorum of servers with that version, restoring this value's consistency.

If a Byzantine client and a colluding proxy try to update only part of the nodes with a certain v, a read may return two kinds of values: (1) If the read quorum will witness v, it will be resolved and propagated to at least a quorum of nodes meaning that v will be written correctly. As a result of this resolve, every following read will return v (or a newer value). (2) If a read will not witness v, the most recent version of a correct write will be returned. Thus, the hardened system protects against such attempts.

Finally, if a Byzantine client is detected by the system administrator and removed, its ACL and certificate can be revoked immediately. This way any potentially future signed writes saved by a colluder will be voided.

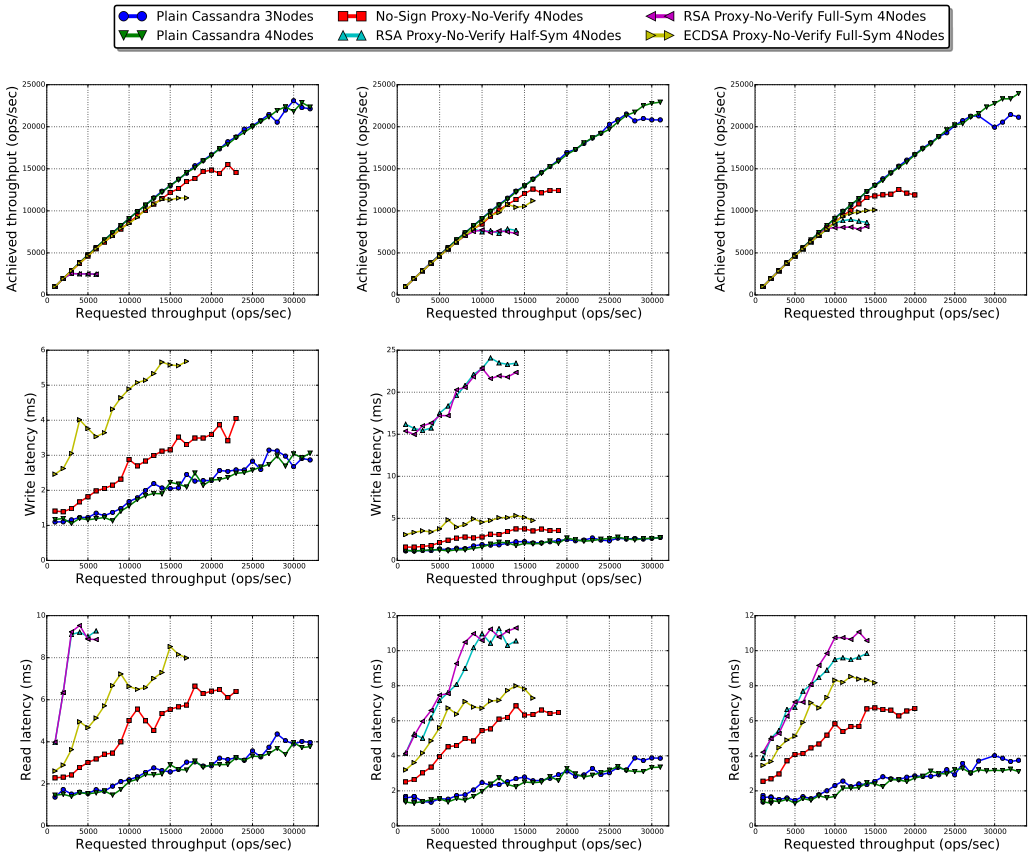
5.5 Deleting Values

In Cassandra, deleting a value is done by replacing it with a *tombstone*. The tombstone is replied to any request for this value to indicate its deletion. Occasionally, a garbage collector removes all tombstones that are older than a configurable time (10 days by default).

Even in a benign environment, a deleted value might reappear, e.g., when a failed node recovers after missing a delete operation and passing all garbage collection intervals in other nodes. Mimicking this, a Byzantine node can ignore delete operations and propagate the deleted values to correct nodes after the garbage collection interval.

We define the following counter-measures: (1) Every delete is signed by a client as in the write operation. This signature is stored in the tombstone. A client completes a delete only after obtaining a Byzantine quorum of signed acknowledgments. (2) During each garbage collection interval, a node must run at least once the anti-entropy tool among a Byzantine quorum of nodes, fetching all missed tombstones. (3) A node will accept writes of values that are not older than the configured time for garbage collection interval. Since the node runs the anti-entropy tool periodically, even if a deleted value is being fetched, the tombstone will overwrite it. (4) A node that receives a store value older than the configured garbage collector time will issue a read for the value and accept it only if a Byzantine quorum approves that the value is live.

We explore a few additional attack vectors, including membership, clocks and network attacks in Appendix B.



(a) Workload A **(b)** Workload B **(c)** Workload C

Figure 5 Comparing the best variants against plain Cassandra and the algorithm with *No-Sign* using workloads A, B and C. In the write latency of (a), we left the RSA variants out as they rapidly grew to ≈65ms latency.

6 Performance

We implemented our algorithms[1] as patches to Cassandra 2.2.4[2]. We evaluated the performance of the variants of our solution and compared them to the original Cassandra using the standard YCSB 0.7[3] benchmark [14], adjusted to use our BFT client library[4]. We used Datastax's Java driver 2.1.8[5] on the client side. There are nearly 390K *LOC* (lines of code) in Cassandra. Our patch added about 3.5K LOC to the servers code and about 4K LOC to the client code (including YCSB integration), which uses the client driver as is. Our entire code adds less than 2% LOC.

All experiments were run on 4 to 5 machines (Ubuntu14, dual 64-bit 6 core 2.2GHz Intel Xeon E5 CPUs, 32GB of RAM, 7200 RPM hard drive and 10Gb ethernet), one for the client and three to four for the Cassandra nodes.

[1] https://github.com/ronili/HardenedCassandra
[2] https://github.com/apache/cassandra/tree/cassandra-2.2.4
[3] https://github.com/brianfrankcooper/YCSB/tree/0.7.0
[4] https://github.com/ronili/HardenedCassandraYCSB
[5] https://github.com/datastax/java-driver/tree/2.1.8

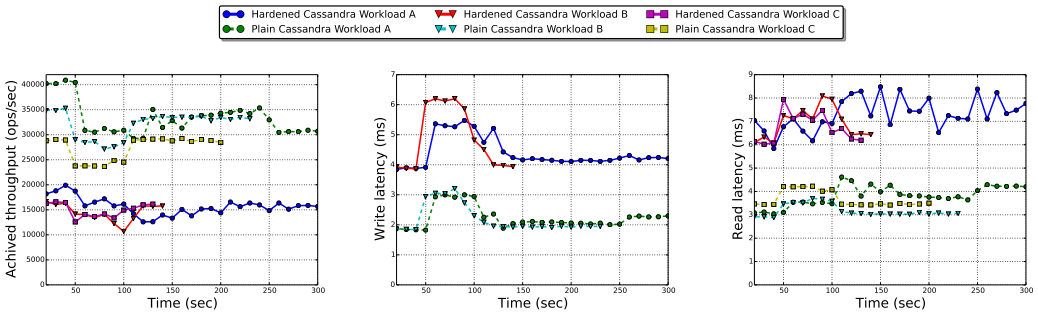


Figure 6 Comparing our best solution (ECDSA Proxy-No-Verify Full-Sym) against plain Cassandra in a benign failure of one node in a 4 nodes setup.

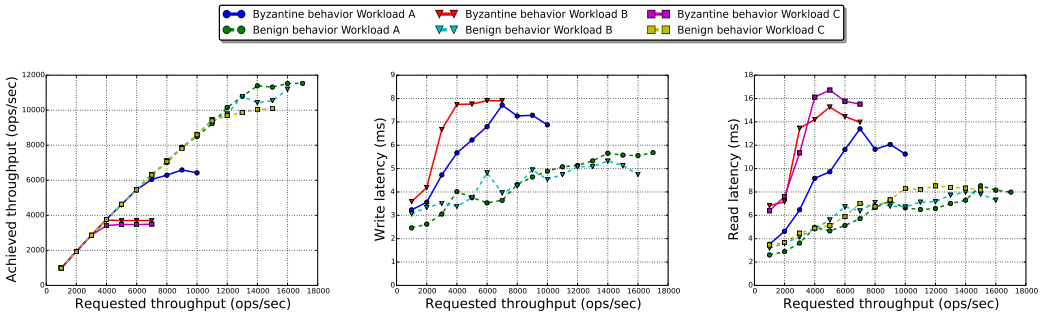


Figure 7 Comparing the best solution in benign behavior and having one node only replying with bad signatures.

We pre-loaded the database with 100,000 rows and benchmarked it with five YCSB workloads as follows: (1) Workload A - 50/50 reads/writes. (2) Workload B - 95/5 reads/writes. (3) Workload C - only reads. (4) Workload D - 95/5 reads/writes, where the reads are for the latest inserts (and not random). (5) Workload F - 50/50 writes/Read-Modify-Writes. In all workloads other than D, the write is for one column while in workload D it is for the entire row. Every workload ran with 100 client threads, performing a total of 100,000 operations with varying throughput targets. The tables consisted of 10 columns (default in YCSB) as well as tables consisting of one value, modeling a key-value datastore. Each value is of size 100 bytes while the key size is randomly chosen in the range of 5-23Bytes. Thus, each record/line with 10 columns has an average length of 1014Bytes. As YCSB throttles the requests rate to the achieved maximum throughput, we run each experiment until obtaining a stable throughput.

We implemented the algorithms of Figures 8 and 9 where the proxy authenticates the acknowledgments; denote these as *Proxy-Verifies*. We also implemented the variant where the proxy does not verify the acknowledgments and lets the client fetch more acknowledgments in case it is not satisfied; denote it *Proxy-No-Verify*. We ran that last algorithm in two modes, one where the proxy is in charge of resolving inconsistent reads, and one where the client is. We present only the former as both behaved similarly.

We analyzed MAC tags in two steps: (1) using MAC tags on messages from nodes to client, denoted *Half-Sym* and (2) using it for both ways, denoted *Full-Sym*.

We used two types of private key signatures: (1) RSA with keys of length of 2048b and (2) ECDSA with keys of length 256b and the *secp256r1* curve. For symmetric keys, we used keys of length of 128b with the *HMAC* MAC algorithm and *SHA256* [20] for hashing.

To evaluate the cost of our algorithm without cryptographic overhead, we ran them also without any signatures. That is, we swapped the signing methods with a *base64* encoded on a single char, referred to as *No-Sign*.

6.1 Performance with No Failures

Figures 5 presents the performance results for the multi-column model for workloads A-C. Workloads D and E are qualitatively similar and can be found in [22]. Our best solution is the variant where the proxy does not verify the acknowledgments, and we use ECDSA and MAC tags for both ways (ECDSA Proxy-No-Verify Full-Sym 4Nodes). The slowdown here is roughly a factor of 2-2.5 in terms of the maximum throughput, 2.5-3 in the write latency and 2-4 in the read latency. The No-Sign experiment represents the BFT algorithmic price including larger quorums, extra verifications and storing signatures. The ECDSA experiment adds the cryptography cost. Notice that RSA has a significant negative performance impact.

We have also studied the performance in the key-value model, i.e., a table with one non-key column. For lack of space, the results are in the full version [22].

6.2 Performance Under Failures

Figure 6 presents the performance of our best solution in a single stopped node scenario. We run workload A on a four nodes setup, with maximum throughput. After 50 seconds, we stopped one node for 30 seconds and then restarted it. It took the node between 20 to 30 seconds to start, after which the other nodes started retransmitting the missed writes to the failed node. In our best solution, the distributed retransmitting took about 250 seconds and in the plain Cassandra, about 170 seconds. We repeated this test with workloads B and C with one change, failing the node in $t = 40$ instead of $t = 50$. In this experiment too our solution performs similarly to plain Cassandra.

In Figure 7, we present the performance of our best solution when one node always returns a bad signature. This impacts the entire system as on every failed signature verification, the client has to contact the proxy again. Further, on every read that addresses the Byzantine node, a resolving and a write-back process is initiated.

We have also explored a stalling proxy attack, where the proxy waits most of the timeout duration before supplying the client with the response. See Appendix C.1.

7 Conclusion

Cassandra's wide adoption makes it a prime vehicle for exploring various aspects of distributed data stores. In our work, we have studied Cassandra's vulnerabilities to Byzantine failures and explored various design alternatives for hardening it against such failures.

We have also evaluated the attainable performance of our design alternatives using the standard YCSB benchmark. The throughput obtained by our best Byzantine tolerant design was only 2-2.5 times lower than plain Cassandra while write and read latencies are only a 2-3X and 2-4X higher, respectively, than in the benign system.

Performance wise, the two most significant design decisions are the specific cryptographic signatures and resolving all conflicts during reads only. Our novel design of sending a vector of MAC tags, signed by itself with the symmetric key of the client and target node, plus the ECDSA public key signature, means that the *usual path* involves no public key verifications and only one elliptic curve signature, but no costly RSA signatures.

References

1 Riak. http://basho.com/products/riak-kv/.

2 Marcos K Aguilera and Ram Swaminathan. Remote storage with byzantine servers. In *Proc. of ACM SPAA*, pages 280–289, 2009.

3 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. on Dependable and Secure Computing*, 8(4):564–577, 2011.

4 Leonardo Aniello, Silvia Bonomi, Marta Breno, and Roberto Baldoni. Assessing Data Availability of Cassandra in the Presence of non-accurate Membership. In *Proc. of the 2nd ACM International Workshop on Dependability Issues in Cloud Computing*, 2013.

5 Apache. Cassandra. http://cassandra.apache.org/.

6 Roberto Baldoni, Marco Platania, Leonardo Querzoni, and Sirio Scipioni. A peer-to-peer filter-based algorithm for internal clock synchronization in presence of corrupted processes. In *Proc. of the IEEE PRDC*, pages 64–72, 2008.

7 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the Annual IEEE/IFIP DSN*, pages 355–362, 2014.

8 Ken Birman and Roy Friedman. Trading Consistency for Availability in Distributed Systems. Technical Report TR96-1579, Cornell University, 1996.

9 Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009.

10 Christian Cachin, Dan Dobre, and Marko Vukolić. Separating data and control: Asynchronous BFT storage with 2t+ 1 data replicas. In *Stabilization, Safety, and Security of Distributed Systems*, pages 1–17. Springer, 2014.

11 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

12 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2), 2008.

13 Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *22nd ACM SOSP*, pages 277–290, 2009.

14 Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the ACM Symposium on Cloud Computing*, pages 143–154, 2010.

15 Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical Hardening of Crash-tolerant Systems. In *Proc. of the USENIX Annual Technical Conference*, ATC, pages 41–41, 2012.

16 Inc DataStax. Apache Cassandra 2.2. http://docs.datastax.com/en/cassandra/2.2/.

17 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM OSR*, 41(6):205–220, 2007.

18 John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

19 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, 1988.

20 D Eastlake and Tony Hansen. US secure hash algorithms (SHA and HMAC-SHA). Technical report, RFC 4634, July, 2006.

21 Christof Fetzer and Flaviu Cristian. Integrating external and internal clock synchronization. *Real-Time Systems*, 12(2):123–171, 1997.

22 Roy Friedman and Roni Licher. Hardening cassandra against byzantine failures. *CoRR*, abs/1610.02885, 2016. URL: http://arxiv.org/abs/1610.02885.

23 Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proc. of the 15th Usenix Conference on File and Storage Technologies*, FAST, pages 149–165, 2017.

24 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proc. of the 5th ACM European Conference on Computer Systems*, pages 363–376, 2010.

25 Maurice Peter Herlihy. Replication methods for abstract data types. Technical report, DTIC Document, 1984.

26 Håvard D Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM TOCS*, 33(2), 2015.

27 Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

28 Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS OSR*, 44(2):35–40, 2010.

29 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

30 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

31 Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *Distributed Computing*, pages 487–489. Springer, 2005.

32 Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM TOCS*, 29(4), 2011.

33 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

34 David Mills, Jim Martin, Jack Burbank, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. *IETF RFC5905, June*, 2010.

35 F. Pedone, N. Schiper, and J. E. Armendáriz-Iñigo. Byzantine Fault-Tolerant Deferred Update Replication. In *Proc. of the 5th Latin-American Symposium on Dependable Computing (LADC)*, pages 7–16, 2011.

36 Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 288–301. ACM, 1997. doi:10.1145/268998.266711.

37 Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

38 Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*, 2006.

39 Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, Petros Maniatis, et al. Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, pages 169–184, 2009.

40 Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems*, pages 261–269. Springer, 2002.

41 Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware*, pages 55–70. Springer, 1998.

A Pseudocode

The pseudocode of the basic hardened write protocol appears in Figure 8 while the basic hardened read protocol is in Figure 9.

```
1:  function NodeWrite(k, v, ts, clientSign, clientID)
2:      if clientSign is valid then
3:          nodeSign ← ComputeSignature(clientSign)
4:                                          ▷ The client signature covers a fresh ts
5:          Store locally < k, v, ts, clientSign, clientID >
6:          return nodeSign                 ▷ A verifiable acknowledgment
7:      end if
8:  end function
9:
10: function ProxyWrite(k, v, ts, clientSign, clientID)
11:     for each node n that is responsible for k do          ▷ N nodes
12:         Send the write request to n
13:     end for
14:     Wait for 2f + 1 verified acknowledgements OR timeout
15:                          ▷ Verified in the manner of correct node signature
16:     return responses
17: end function
18:
19: function ClientWrite(k, v)
20:     ts ← Current timestamp              ▷ From a secure synchronized clock
21:     clientSign ← ComputeSignature(k || v || ts)
22:     p ← Some random system node
23:     Send write with <k, v, ts, clientSign, clientID> to p
24:     Wait for acknowledgments OR timeout
25:     if |validResponses| ≥ 2f + 1 then
26:         return Success
27:     end if
28:     if p = ⊥ OR contactedNodes > f  then
29:         return Failure
30:     end if
31:     goto line 22                        ▷ Use another node as proxy
32: end function
```

Figure 8 Our hardened write algorithm. ClientWrite is invoked on the client for each write. ProxyWrite is invoked on the proxy node by the client. NodeWrite is invoked on a node that has the responsibility to store the value. *Store locally* appends the write to an append log without any read. When k is queried, the latest store (by timestamp) is retrieved.

B Additional Attack Vectors

B.1 Column Families vs. Key-Value semantics

The algorithms described so far reflect only key-value semantics. Yet, we also supports Cassandra's column family semantics. In the latter, a client signs each column separately, producing a number of signatures that is equivalent to the number of non-key columns. This is needed for reconciling partial columns writes correctly according to Cassandra's semantics. For example, consider a scheme with two non-key columns A and B. One node can hold an updated version of A and a stale version of B while another node might hold the opposite state. A correct read should return one row containing the latest columns for both.

Nodes acknowledgments can still include only a single signature covering all columns. This is because the purpose of signatures here is to acknowledge the operation.

B.2 Membership View

The membership implementation of Cassandra is not Byzantine proof as faulty nodes can temper other's views by sending false data [4]. In addition, Byzantine seed nodes can partition the system into multiple subsystems that do not know about each other. This is by exposing different sets of nodes to different nodes.

To overcome this, each node's installation should be signed by the trusted system administrator with a logical timestamp. The logical timestamp enables nodes to verify they are using an updated configuration. Each node must contact at least $f + 1$ seed nodes in order to get at least one correct view. This requires the system administrator to pre-configure

```
1:  function NodeRead(k, clientTs)
2:      if k is stored in the node then
3:          < v, ts, clientSign, clientID >← Newest (timestamp manner) stored timestamp, value and client
    signature with k
4:      else
5:          clientSign ← EMPTY
6:      end if
7:      nodeSign ← ComputeSignature(k||h(v)||clientSign||clientTs)
8:      if isDigestQuery then
9:          return < h(v), ts, clientSign, clientID, nodeSign >
10:                                                          ▷ The hash is matched in the proxy
11:     else
12:         return < v, ts, clientSign, clientID, nodeSign >
13:     end if
14: end function
15:
16: function ProxyRead(k, clientTs)
17:     targetEndpoints ← allRelevantNodes for k OR a subset of 2f + 1 fastest relevant nodes ▷ Optimization
18:     dataEndpoind ← One node from targetEndpoints
19:     Send read data to dataEndpoind
20:     Send read digest to targetEndpoints \ {dataEndpoind}
21:     Wait for 2f + 1 verified responses OR timeout
22:     if timeout AND all nodes were targeted in line 17 then
23:         return ⊥
24:     end if
25:     if got data response AND all digests match then
26:         return < v, nodesSign >
27:     end if
28:     Send read data to all nodes in allRelevantNodes                                  ▷ N nodes
29:     Wait for 2f + 1 verified responses OR timeout
30:     if timeout then
31:         return ⊥
32:     end if
33:     resolved ← Latest v in responses that is clientSign verified.
34:     Write-back resolved to allRelevantNodes
35:                                                  ▷ except those that are known to be updated
36:     Wait to have knowledge about 2f + 1 verified updated nodes OR timeout ▷ Returned updated data or a
    write back ACK
37:     if timeout then
38:         return ⊥
39:     end if
40:     return < resolved, nodesSigns, allNodesValues >
41: end function
42:
43: function ClientRead(k)
44:     clientTs ← Current timestamp                                          ▷ Fresh timestamp
45:     p ← Some random system node
46:     Send read request with < k, clientTs > to p
47:     Wait for responses OR timeout
48:     if |validNodesSign| ≥ 2f + 1 then
49:         ▷ If write-back is observed, the resolved row is verified with the original read answers to ensure it was
    required
50:         return data
51:     end if
52:     p ← Random node that was not used in this read as a proxy
53:     if p = ⊥ OR contactedNodes > f  then
54:         return Failure
55:     end if
56:     goto line 46
57: end function
```

Figure 9 Our hardened read algorithm. ClientRead is invoked by the client for each read. ProxyRead is invoked on the proxy by the client. NodeRead is invoked on a node that is responsible to store the value.

manually the first $f + 1$ nodes view as they cannot trust the rest. Let us emphasize that Byzantine seeds cannot forge false configurations. Rather, they can only hide configurations by not publishing them.

Here, we adopt the work on BFT push-pull gossip by Aniello et al. [4]. Their solution solves the dissemination issues by using signatures on the gossiped data.

B.3 Synchronized Clock

In plain Cassandra, as well as in our case, each write includes a wall-clock timestamp used to order the writes. With this, strong consistency cannot be promised unless local clocks are perfectly synchronized. For example, consider two clients that suffer from a clock skew of Δ. If both clients write to the same object in a period that is shorter than Δ, the later write might be attached with a smaller timestamp, i.e., the older write wins.

In a benign environment, when ensuring a very low clock skew, for most applications, these writes can be considered as parallel writes so any ordering of them is correct. For time synchronization, Cassandra requires the administrator to provide an external solution such as NTP. In our work, we follow this guideline using the latest version of NTP that can tolerate Byzantine faults when ensuring the usage of SSL and authentication measures [6, 34]. We configure this service so that all servers could use it as is and clients would be able only to query it, without affecting the time.

Alternatively, one could use external clocks such as GPS clocks, atomic clocks or equivalents [21], assuming Byzantine nodes can neither control them nor the interaction with them. Finally, Cassandra nodes can ignore writes with timestamps that are too far into the future to be the result of a normal clock's skew.

B.4 Other Network Attacks

Cassandra might suffer known overlay networks attacks, such as *Sybil attacks* [18] and *Eclipse attacks* [38]. In a Sybil attack, attackers create multiple false entities. In Cassandra, multiple node ids may lead to the same node, thereby fooling a client into storing its data only on a single Byzantine replica. As suggested in [18], here we rely on a trusted system administrator to be the sole entity for approving new entities that can be verified using PKI.

In an Eclipse attack, attackers divert requests towards malicious entities. E.g., a proxy might try to target only Byzantine replicas. To overcome this, clients request verifiable acknowledgments and count the number of correct repliers. If a proxy fails to provide these, alternative proxies are contacted until enough correct nodes have been contacted. Additionally, Section 5.3.1 explains how we handle a proxy that diverts requests to irrelevant nodes.

C Additional Performance Issues

C.1 Stalling Proxy

In the stalling proxy performance attack, following a correct execution of an operation, the proxy waits most of the timeout duration before supplying the client with the response. Consequently, the system's performance might decrease dramatically. Since the attack effects vary depending on the timeout configuration, the attack can be mitigated by lowering the timeout as low as possible. On the contrary, a tight timeout might fail correct requests during congestion times. The right optimization of timeouts relies on several deployment factors, e.g., the application requirements, the connection path of the client to system, the network topology of the nodes and more. Therefore, we have no definitive insights when facing this case. Finally, we would like to point out that the client can be configured to contact the fastest responding nodes first and thus reduce the effect of this attack.

Vulnerability-Tolerant Transport Layer Security*

André Joaquim[1], Miguel L. Pardal[2], and Miguel Correia[3]

1 **INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal**
andre.joaquim@tecnico.ulisboa.pt
2 **INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal**
miguel.pardal@tecnico.ulisboa.pt
3 **INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal**
miguel.p.correia@tecnico.ulisboa.pt

Abstract

SSL/TLS communication channels play a very important role in Internet security, including cloud computing and server infrastructures. There are often concerns about the strength of the encryption mechanisms used in TLS channels. Vulnerabilities can lead to some of the cipher suites once thought to be secure to become insecure and no longer recommended for use or in urgent need of a software update. However, the deprecation/update process is very slow and weeks or months can go by before most web servers and clients are protected, and some servers and clients may never be updated. In the meantime, the communications are at risk of being intercepted and tampered by attackers.

In this paper we propose an alternative to TLS to mitigate the problem of secure communication channels being susceptible to attacks due to unexpected vulnerabilities in its mechanisms. Our solution, called Vulnerability-Tolerant Transport Layer Security (VTTLS), is based on diversity and redundancy of cryptographic mechanisms and certificates to ensure a secure communication even when one or more mechanisms are vulnerable. Our solution relies on a combination of k cipher suites which ensure that even if $k-1$ cipher suites are insecure or vulnerable, the remaining cipher suite keeps the communication channel secure. The performance and cost of VTTLS were evaluated and compared with OpenSSL, one of the most widely used implementations of TLS.

1998 ACM Subject Classification C.2.2 Network Protocols, D.4.6 Security and Protection

Keywords and phrases Secure communication channels, Transport layer security, SSL/TLS, Diversity, Redundancy, Vulnerability tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.28

1 Introduction

Secure communication protocols are extremely important in the Internet. *Transport Layer Security* (TLS) alone is responsible for protecting most economic transactions done using the Internet, with a value too high to estimate. These protocols allow entities to exchange messages or data over a secure channel in the Internet. A secure communication channel has three main properties: *authenticity* – no one can impersonate the sender; *confidentiality* – only the intended receiver of the message is able to read it; and *integrity* – tampered messages can be detected.

* This work was supported by the European Commission through project H2020-653884 (SafeCloud) and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID)

Several secure communication channel protocols exist nowadays, with different purposes but with the same goal of securing communication. TLS is a widely used secure channel protocol. Originally called Secure Sockets Layer (SSL), its first version was SSL 2.0, released in 1995. SSL 3.0 was released in 1996, bringing improvements to its predecessor such as allowing forward secrecy and supporting SHA-1. Defined in 1999, TLS did not introduce major changes in relation to SSL 3.0. TLS 1.1 and TLS 1.2 are upgrades to TLS 1.0 which brought improvements such as mitigation of cipher block chaining (CBC) attacks and supporting more block cipher modes to use with AES.

Other widely used secure channel protocols are IPsec and SSH. *Internet Protocol Security* (IPsec) is a network layer protocol that protects the communication at a lower level than SSL/TLS, which operates at the transport layer [18]. IPsec is an extension of the Internet Protocol (IP) that contains two sub-protocols: AH (Authentication Header) that can assure full packet authenticity and ESP (Encapsulating Security Payload) that can protect confidentiality and integrity of the payload of the packets. *Secure Shell* (SSH) is an application-layer protocol used for secure remote login and other secure network services over an insecure network [38].

A secure channel protocol becomes insecure when a vulnerability is discovered. Vulnerabilities may concern the specification of the protocol, the cryptographic mechanisms used, or specific implementations of the protocol. Many vulnerabilities have been discovered in TLS originating new versions of the protocol, deprecating cryptographic mechanisms or enforcing additional security measures. Concrete implementations of TLS have been also found vulnerable due to implementation bugs. The processes of deprecation or software update are very slow and can take a long time to become effective [2]. Also, they may not even reach all the affected servers and clients. This means that the communications between devices are at risk of interception or tampering by attackers for a long period.

This paper proposes a *vulnerability-tolerant* communication channel protocol, based on TLS, named Vulnerability-Tolerant Transport Layer Security (vtTLS). A vulnerability-tolerant channel is characterized by not relying on individual cryptographic mechanisms, so that if any one of them is found vulnerable, the channel still remains secure. The idea is to leverage *diversity* and *redundancy* of cryptographic mechanisms and keys by using more than one set of mechanisms/keys. This use of diversity and redundancy is inspired in previous works on intrusion tolerance [34], diversity in security [21, 13] and moving-target defenses [7].

Consider for example SHA-1 and SHA-3, two hash functions that may be used to generate message digests. If used in combination and SHA-1 eventually becomes insecure, vtTLS would rely upon SHA-3 to keep the communication secure.

vtTLS is configured with a parameter k ($k > 1$), the *diversity factor*, that indicates the number of different cipher suites and different mechanisms for key exchange, authentication, encryption, and signing. This parameter means also that vtTLS remains secure as long as less than k vulnerabilities exist. As vulnerabilities and, more importantly, zero-day vulnerabilities cannot be removed as they are unknown [4], do not appear in large numbers in the same components, we expect k to be usually small, e.g., $k = 2$ or $k = 3$.

Although TLS supports strong encryption mechanisms such as AES and RSA, there are factors beyond mathematical complexity that can contribute to vulnerabilities. Diversifying encryption mechanisms includes diversifying certificates and consequently keys (public, private, shared). Diversity of certificates is a direct consequence of diversifying encryption mechanisms due to the fact that each certificate is related to an authentication and key exchange mechanism.

The main contribution of this paper is vtTLS, a new protocol for secure communication channels that uses diversity and redundancy to tolerate vulnerabilities in cryptographic mechanisms. The experimental evaluation shows that vtTLS has an acceptable overhead in relation to the TLS implementation in OpenSSL v1.0.2g [35].

The rest of the document is organized as follows. Section 2 presents background and related work. Section 3 presents the protocol and Section 4 its implementation. Section 5 presents the experimental evaluation and Section 6 concludes the paper.

2 Background and Related Work

This section presents related work on diversity (and redundancy) in security, provides background information on TLS, and discusses vulnerabilities in cryptographic mechanisms and protocols.

2.1 Diversity

The term *diversity* is used to describe multiple version software in which redundant versions are deliberately created and made different between themselves [21]. Without diversity, all instances are the same, with the same implementation vulnerabilities. Using diversity it is possible to present the attacker with different versions, hopefully with different vulnerabilities. Software diversity targets mostly software implementation and the ability of the attacker to replicate the user's environment. Diversity does not change the program's logic, so it is not helpful if a program is badly designed. According to Littlewood and Strigini, multi-version systems on average are more reliable than those with a single version [21]. They also state that the key to achieve effective diversity is to make the dependence between the different programs as low as possible. Therefore, attention is needed when choosing the diverse versions. The trade-off between individual quality and dependence needs to be assessed and evaluated, as it impacts the correlation between version failures.

Recently there has been some discussion on the need for moving-target defenses. Such defenses dynamically alter properties of programs and systems in order to overcome vulnerabilities that eventually appear in static defense mechanisms [11]. There are two types of moving-target defenses: proactive and reactive [7]. Proactive defenses are generally slower than reactive defenses as they prevent attacks by increasing the system complexity periodically. Reactive defenses are faster as they are activated when they receive a trigger from the system when an attack is detected. This may cause a problem where an attack is performed but not detected. In this case, reactive defenses are worthless, but proactive defenses may prevent that attack from being successful. The best approach would be to implement both [30]. Nevertheless, these defences are as good as their ability to make an unpredictable change to the system.

Earlier, Avižienis and Chen introduced N-version programming (NVP) [3]. NVP is defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. The authors state that in order to use redundant programs to achieve fault tolerance the redundant program must contain independently developed alternatives routines for the same functions. The N in NVP comes from N diverse versions of a program developed by N different programmers, that do not interact with each other regarding the programming process. One of the limitations of NVP is that every version is originated from the same initial specification. There is the need to assure the initial specification's correctness, completeness and unambiguity prior to the versions development.

There is some work on obtaining diversity without explicitly developing N versions [20]. Garcia *et al.* show that there is enough diversity among operating systems for several practical purposes [13]. Homescu *et al.* use profile-guided optimization for automated software diversity generation [15]. Transparent Runtime Randomization (TRR) dynamically and randomly relocates parts of the program to provide different versions [37]. Proactive obfuscation aims to generate replicas with different vulnerabilities [26]. We introduced the idea of using diversity for vulnerability-tolerant channels in a short (4-page) paper [16]. The present paper greatly expands that previous work.

2.2 TLS Protocol

TLS has two main sub-protocols: the Handshake Protocol and the Record Protocol.

The Handshake Protocol is used to establish (or resume) a secure session between two communicating parties – a client and a server. A session is established in several steps, each one corresponding to a different message.

The Record protocol is the sub-protocol which processes the messages to send and receive after the handshake, i.e., in normal operation. Regarding an outgoing message, the first operation performed by the Record protocol is fragmentation. After fragmenting the message, each block may be optionally compressed, using the compression method defined in the Handshake. Each potentially compressed block is now transformed into a ciphertext block by encryption and message authentication code (MAC) functions. Each ciphertext block contains the protocol version, content type and the encrypted form of the compressed fragment of application data, with the MAC, and the fragment's length. When using block ciphers, padding and its length is added to the block. The padding is added in order to force the length of the fragment to be a multiple of the block cipher's block length. When using AEAD ciphers (MAC-then-Encrypt mode using a SHA-2 variant), no MAC key is used. The message is then sent to its destination. Regarding an incoming message, the process is the inverse. The message is decrypted, verified, optionally decompressed, reassembled and delivered to the application.

2.3 TLS Vulnerabilities

We now discuss some of the vulnerabilities discovered in TLS in the past to show the relevance of our work to bring added security to communications. TLS vulnerabilities can be classified in two types: specification and implementation. *Specification vulnerabilities* concern the protocol itself. A specification vulnerability can only be fixed by a new protocol version or an extension. *Implementation vulnerabilities* exist in the code of an implementation of TLS, such as OpenSSL. This section presents some of the most recent [28].

An example attack that exploits a specification vulnerability is *Logjam* [1]. The attack consists in exploiting several weak parameters in the Diffie-Hellman key exchange. Logjam is a man-in-the-middle attack that downgrades the connection to a weakened Diffie-Hellman mode. This man-in-the-middle attack changes the cipher suites used in the `DHE_EXPORT` cipher suite, forcing the use of weaker Diffie-Hellman key exchange parameters. As the server supports this valid Diffie-Hellman mode, the handshake proceeds without the server noticing the attack. The server proceeds to compute its premaster secret using weakened Diffie-Hellman parameters. The client sees that the server has chosen a seemingly normal DHE option and proceeds to compute its secret also with weak parameters. At this point, the man-in-the-middle can use the precomputation results to break one of the secrets and establish the connection to the client pretending to be the server.

Heartbleed is an example of an *implementation vulnerability*. It was a bug in C code that existed in OpenSSL 1.0.1 through 1.0.1f, when the heartbeat extension was introduced and enabled by default [27]. The Heartbleed vulnerability allowed an attacker to perform a buffer over-read, reading up to 64 KB from the memory of the victim [6].

2.4 Vulnerabilities in Cryptographic Schemes

This section presents example vulnerabilities in cryptographic mechanisms used by TLS.

2.4.1 Public-key Cryptography

RSA is a widely-used public-key cryptography scheme. Its security is based on the difficulty of factorization of large integers and the RSA problem [23]. RSA can be considered to be broken if these problems can be solved in a practical amount of time.

Kleinjung *et al.* performed the factorization of RSA-768, a RSA number with 232 digits [19]. The researchers state they spent almost two years in the whole process, which is clearly a non-practical time. Factorizing a large integer is different from breaking RSA, which is still secure. As of 2010, these researchers concluded that RSA-1024 would be factored within five years. As for now, no factorization of RSA-1024 has been publicly announced, but key sizes of 2048 and 3072 bits are now recommended [10].

Shor designed a quantum computing algorithm to factorize integers in polynomial time [29]. However, it requires a quantum computer able to run it, which is still not publicly available.

2.4.2 Symmetric Encryption

The Advanced Encryption Standard (AES), originally called Rijndael, is the current American standard for symmetric encryption [25]. AES can be employed with different key sizes – 128, 192 or 256 bits. The number of rounds corresponding to each key size is, respectively, 10, 12 and 14. AES is used by many protocols, including TLS.

The most successful cryptanalysis of AES was published by Bogdanov *et al.* in 2011, using a biclique attack, a variant of the MITM attack [5]. This attack achieved a complexity of $2^{126.1}$ for the full AES with 128-bit (AES-128). The key is therefore reduced to 126-bit from the original 128-bit, but it would still take many years to successfully attack AES-128. Ferguson *et al.* presented the first known attacks on the first seven and eight rounds of Rijndael [12]. Although it shows some advance in breaking AES, AES with a key of 128 bits has 10 rounds.

2.4.3 Hash Functions

The main uses for hash functions are data integrity and message authentication. A hash, also called message digest or digital fingerprint, is a compact representation of the input and can be used to uniquely identify that same input [22]. If a hash function is not collision-resistant, it is vulnerable to collision attacks. Some generic attacks to hash function include brute force attacks, birthday attacks and side-channel attacks.

The Secure Hash Algorithm 1 (SHA-1) is a cryptographic hash function that produces a 160-bit message digest. Its use is not recommended for some years [10], although the first collision was discovered only recently [32]. There have been some previous attacks against SHA-1. Stevens *et al.* presented a freestart collision attack for SHA-1's internal compression function [33]. Taking into consideration the Damgard-Merkle [24] construction for hash

functions and the input of the compression function, a freestart collision attack is a collision attack where the attacker can choose the initial chaining value, also known as initialisation vector (IV). Freestart collision attacks being successful does not imply that SHA-1 is insecure, but it is a step forward in that direction.

In 2005, Wang *et al.* presented a collision attack on SHA-1 that reduced the number of calculations needed to find collisions from 2^{80} to 2^{69} [36]. The researchers claim that this was the first collision attack on the full 80-step SHA-1 with complexity inferior to the 2^{80} theoretical bound. By the year 2011, Stevens improved the number of calculations needed to produce a collision from 2^{69} to a number between $2^{60.3}$ and $2^{65.3}$ [31].

3 Vulnerability-Tolerant TLS

vtTLS is a new protocol that provides vulnerability-tolerant secure communication channels. It aims at increasing security using diverse and redundant cryptographic mechanisms and certificates. It is based on the TLS protocol. The protocol aims to solve the main problem originated by having only one cipher suite negotiated between client and server: when one of the cipher suite's mechanisms becomes insecure, the communication channels using that cipher suite may become vulnerable. Although most cipher suites' cryptographic mechanisms supported by TLS 1.2 are believed to be secure, Section 2 shows clearly that new vulnerabilities may be discovered.

Unlike TLS, a vtTLS communication channel does not rely on only one cipher suite. vtTLS negotiates more than one cipher suite between client and server and, consequently, more than one cryptographic mechanism will be used for each *phase*: key exchange, authentication, encryption and MAC. Diversity and redundancy appear firstly in vtTLS in the Handshake protocol, in which client and server negotiate k cipher suites to secure the communication, with $k > 1$.

The strength of vtTLS resides in the fact that even when $(k-1)$ cipher suites become insecure, e.g., because $(k-1)$ of the cryptographic mechanisms are vulnerable, the protocol remains secure. The server chooses the best combination of k cipher suites according to the cipher suites server and client have available. However, the choice of the cipher suites might be conditioned by the certificates of both server and client. vtTLS uses a subset of the k cipher suites agreed-upon in the Handshake Protocol to encrypt the messages.

3.1 Protocol Specification

The vtTLS Handshake Protocol is similar to the TLS Handshake Protocol. We use the same names for the messages in order to help the reader familiarized with TLS.

The messages that require diversity are ClientHello, ServerHello, k-ServerKeyExchange, Server and Client Certificate, and k-ClientKeyExchange.

The first message to be sent is ClientHello to inform the server that the client wants to establish a secure channel for communication.

The server responds with a ServerHello message. This is where the server sends to the client the k cipher suites to be used in the communication. The server also sends its protocol version, a Random structure identical to the one received from the client, the session identifier, and the k cipher suites chosen by the server from the list the client sent.

The server proceeds to send a Server Certificate message containing its k certificates to the client. The k chosen cipher suites are dependent from the server's certificates. Each certificate is associated with one key exchange mechanism (KEM). Therefore, the k cipher suites must use the key exchange mechanisms supported by the server's certificates.

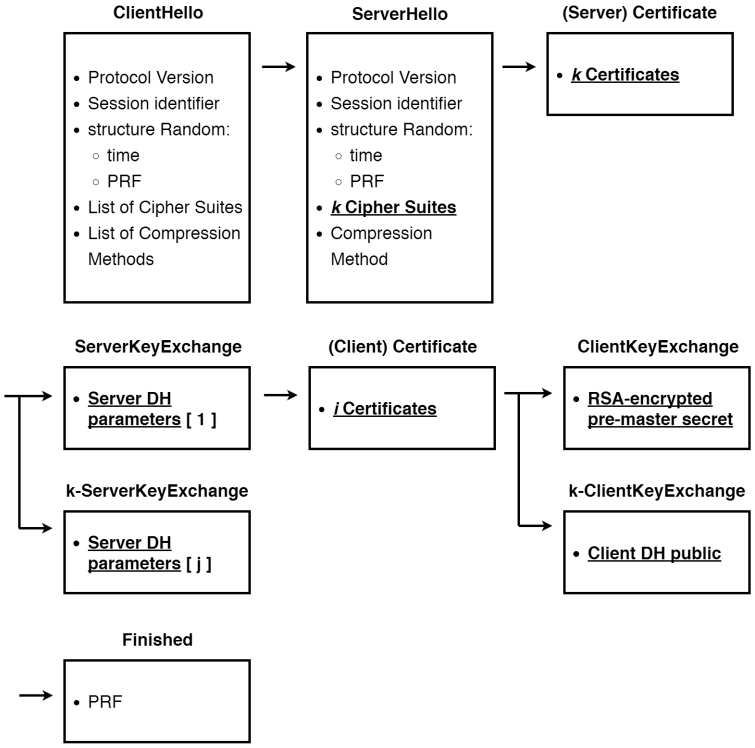


Figure 1 vtTLS Handshake messages with diversity factor k. The places where diversity and redundancy are introduced are marked in bold and underlined.

vtTLS behaves correctly if the server has c certificates, with $0 < c \leq k$. The cipher suites to be used are chosen considering the available certificates. If $c < k$, the diversity is not fully achieved due to the fact that a number of cipher suites will share the same key exchange and authentication mechanisms.

The ServerKeyExchange message is the next message to be sent to the client by the server. This message is only sent if one of the k cipher suites includes a key exchange mechanism like ECDHE or DHE that uses ephemeral keys, i.e., that generate new keys for every key exchange. The contents of this message are the server's DH ephemeral parameters. For every other $k - 1$ cipher suites using ECDHE or DHE, the server sends additional ServerKeyExchange messages with additional diverse DH ephemeral parameters. Instead of computing all the ephemeral parameters and sending them all on a single larger message, the server, after computing one parameter, sends it immediately, sending each parameter in a separate message.

The remaining messages sent by the server to the client at this point of the negotiation, CertificateRequest and ServerHelloDone, are identical to those in TLS 1.2 [9].

The client proceeds to send a (Client) Certificate message containing its i certificates to the server, analogous to the (Server) Certificate message the client received previously from the server.

After sending its certificates, the client sends k ClientKeyExchange messages to the server. The content of these messages is based on the k cipher suites chosen. If m of the cipher suites use RSA as KEM, the client sends m messages, each one with a RSA-encrypted pre-master secret to the server ($0 \leq m \leq k$). If j of the cipher suites use ECDHE or DHE, the

client sends j messages to the server containing its j Diffie-Hellman public values ($0 \leq j \leq k$). Even if a subset of the k cipher suites share the same KEM, this methodology still applies as we introduce diversity by using different parameters for each cipher suite being used.

The server needs to verify the client's i certificates. The client digitally signs all the previous handshake messages and sends them to the server for verification.

Client and server now exchange ChangeCipherSpec messages, like in the Cipher Spec Protocol of TLS 1.2, in order to state that they are now using the previously negotiated cipher suites for exchanging messages in a secure fashion.

In order to finish the Handshake, the client and server send each other a Finished message. This is the first message sent encrypted using the k cipher suites negotiated earlier. Its purpose is for each party to receive and validate the data received in this message. If the data is valid, client and server can now exchange messages over the communication channel.

3.2 Combining Diverse Cipher Suites

Diversity between cryptographic mechanisms can be taken in a soft sense as the use of different mechanisms, or in a hard sense as the use of mechanisms that do not share common vulnerabilities (e.g., because they are based on different mathematical problems). In vtTLS we are interested in using strong diversity in order to claim that no common vulnerabilities will appear in different mechanisms. Measuring the level of diversity is not simple, so we leverage previous research by Carvalho on heuristics for comparing diversity among cryptographic mechanisms [8]. Moreover, not all cryptographic mechanisms can be used together in the context of TLS 1.2 and other security protocols. Here we consider only the combinations of two algorithms, i.e., $k = 2$, for simplicity.

After comparing several hash functions, Carvalho concluded that the best three combinations are the following:

- SHA-1 + SHA-3: not possible in vtTLS as SHA-1 is not recommended and TLS 1.2 does not support SHA-3;
- SHA-1 + Whirlpool: not possible in vtTLS as SHA-1 is not recommended and TLS 1.2 does not support Whirlpool;
- SHA-2 + SHA-3: also not possible in vtTLS as TLS 1.2 does not support SHA-3.

All the remaining combinations suggested in that work cannot also be used because TLS 1.2 does not support SHA-3. All vtTLS cipher suites use either AEAD or SHA-2 (SHA-256 or SHA-384). Having a small range of available hash functions limits the maximum diversity factor achievable concerning hash functions. In a near future, it is expected that a new TLS protocol version supports SHA-3 and makes possible the use of diverse hash functions. Nevertheless, it still possible to achieve diversity by using different variants of SHA-2: SHA-256 and SHA-384.

After comparing several public-key encryption mechanisms, Carvalho concluded that the best four combinations are:

- DSA + RSA: possible as TLS 1.2 supports both functions for *authentication*. However, TLS 1.2 specific cipher suites only support DSA with elliptic curves (ECDSA);
- DSA + Rabin-Williams: not possible as TLS 1.2 does not support Rabin-Williams;
- RSA + ECDH: possible as TLS 1.2 supports both functions for *key exchange*;
- RSA + ECDSA: possible as TLS 1.2 supports both functions for *authentication*.

Regarding authentication, although DSA + RSA is stated as the most diverse combination, TLS 1.2 preferred cipher suites use ECDSA instead of DSA. Using elliptic curves results in a

faster computation and lower power consumption [14]. With that being said, the preferred combination for authentication is RSA + ECDSA.

Regarding key exchange, the most diverse combination is RSA + ECDH. However, in order to grant perfect forward secrecy, the ECDH with ephemeral keys (ECDHE) has to be employed. Concluding, the preferred combination for key exchange is RSA + ECDHE.

The study in [8] did not present any conclusions regarding symmetric-key encryption. However, both AES and Camellia are supported by TLS 1.2 and are considered secure. The most diverse combination is AES256-GCM + CAMELLIA128-CBC: the origin of the two algorithms is different, they were first published in different years, they both have semantic security (as they both use initialization vectors) and the mode of operation is also different. One constraint of using this combination is that there is no cipher suite that uses RSA for key exchange, Camellia for encryption and a SHA-2 variant for MAC. Although RFC 6367 [17] describes the support for Camellia HMAC-based cipher suites, extending TLS 1.2, these cipher suites are not supported by OpenSSL 1.0.2g. Using a cipher suite that uses Camellia, in order to maximize diversity, implies using also SHA-1 for MAC and not using ECDHE for key exchange nor ECDSA for authentication in that cipher suite. Concluding, using Camellia increases diversity in encryption but reduces security in MAC, forcing the use of an insecure algorithm. Nevertheless, diversity in encryption is still an objective to accomplish. We decided that the best option is:

- AES256-GCM + AES128: possible as TLS 1.2 supports both functions.

These functions are, in theory, the same, but employed with a different strength size and mode of operation, they can be considered diverse, although they have an inferior degree of diversity comparing to any of the combinations above.

Concluding, the best combination of cipher suites is arguably:

- `TLS_ECDHE_ECDSA_WITH_AES_ 256_GCM_SHA384` and
- `TLS_RSA_WITH_AES_128_CBC_SHA256`

For key exchange, vtTLS will use Ephemeral ECDH (ECDHE) and RSA; for authentication, it will use Elliptic Curve DSA (ECDSA) and RSA; for encryption, it will use AES-256 with Galois/Counter mode (GCM) and AES-128 with cipher block chaining (CBC) mode; finally, for MAC, it will use SHA-2 variants (SHA-384 and SHA-256). Using this combination of cipher suites, the lowest diversity is with the MAC, due to the fact that TLS 1.2 does not support SHA-3 for now.

4 Implementation

Our implementation of vtTLS was obtained by modifying OpenSSL version 1.0.2g.[1]. Implementing a vtTLS from scratch would be a bad option as it might lead to the creation of vulnerabilities; existing software such as OpenSSL has the advantage of being extensively debugged, although serious vulnerabilities like Heartbleed still appear from time to time. Furthermore, creating a new secure communication protocol, and consequently a new API, would create adoption barriers to programmers otherwise willing to use our protocol. Therefore, we chose to implement vtTLS based on OpenSSL, keeping the same API as far as possible. Although being based on OpenSSL, vtTLS is not fully compatible with it due to its diversity and redundancy features. It is noteworthy that OpenSSL is a huge code base (438,841 lines of code in version 1.0.2g) so modifying it to support diversity has been a considerable engineering challenge.

[1] `https://www.openssl.org`

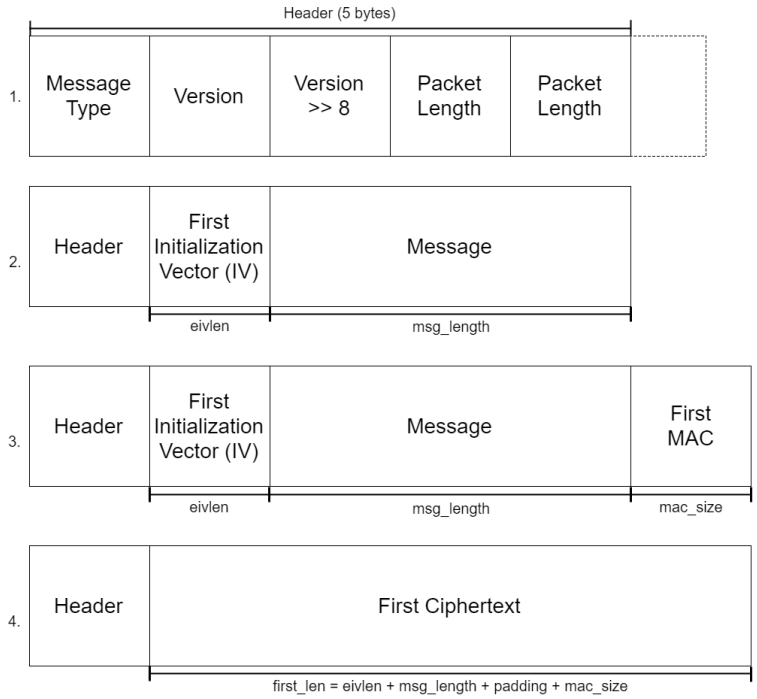


Figure 2 First four steps in the creation of a data message in vtTLS with $k = 2$: first encryption and MAC.

In order to establish a vtTLS communication channel, additional functions are required to fulfill the requirements of vtTLS, such as loading two certificates and corresponding private keys. These functions have a similar name of the ones belonging to the OpenSSL API, to reduce the learning curve. The most relevant functions regarding the setup of the channel are the functions that allow to load the second certificate and private key and allow to check if the second private key corresponds to the second certificate.

Regarding the *Handshake Protocol*, we opted for sending k ServerKeyExchange and ClientKeyExchange messages instead of sending one single ServerKeyExchange and one single ClientKeyExchange, each one with several parameters. This is due to the fact that it makes the code easier to understand and to maintain. If k needs to be increased, it is just needed to send an additional message instead of changing the code related to sending and retrieving ServerKeyExchange and ClientKeyExchange messages.

The encryption and signing ordering is also important in vtTLS. Figure 2 shows the initial steps of the creation of a data message. The figure considers $k = 2$, but shows only the steps regarding the first encryption and the first signature.

Figure 3 shows the final steps of the preparation of a vtTLS message. We opted for maintaining the creation of the MAC prior to the encryption. Using this approach, both message and MACs are encrypted with both ciphers. In this case there is no chance that both MACs are identical, if the hash function used is secure (SHA-2 is considered secure).

The whole sequence is the following:

- Apply the first MAC to the plaintext message;
- Encrypt the original message and its MAC with the first encryption function;
- Apply the second MAC to the first ciphertext;
- Encrypt the first ciphertext and its MAC with the second encryption function.

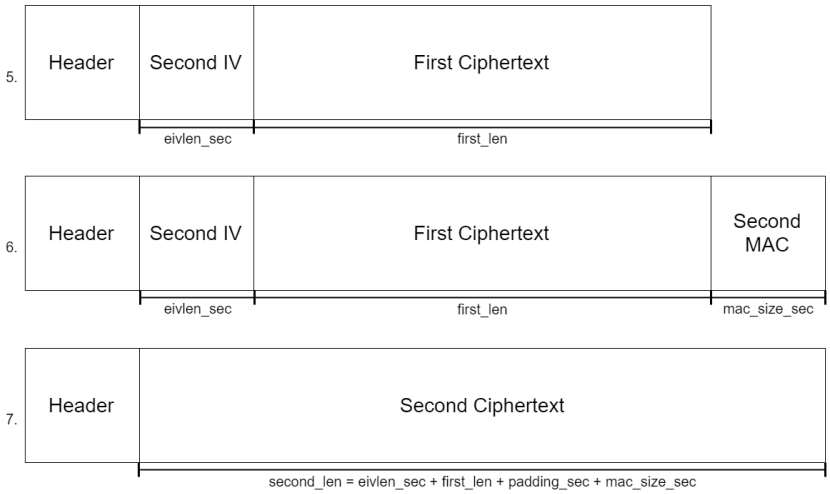


Figure 3 Remaining three steps in the creation of a data message in vtTLS with $k = 2$: second encryption and MAC.

In relation to the *Record Protocol*, signing and encrypting k times has a cost in terms of message size. Figures 2 and 3 show also the expected increase of the message size due to the use of a second MAC and a second encryption function (for $k = 2$). For TLS 1.2 (OpenSSL), the expected size of a message is $first\_len = eivlen + msg\_length + padding + mac\_size$, where *eivlen* is the size of the initialization vector (IV), *msg_length* the original message size, *padding* the size of the padding in case a block cipher is used, and *mac_size* the size of the MAC (Figure 2). For vtTLS, the additional size of the message is $eivlen\_sec + first\_len + padding\_sec + mac\_size\_sec$, where *eivlen_sec* is the size of the IV associated with the second cipher and *mac_size_sec* the size of the second MAC.

In the best case, the number of packets is the same for OpenSSL and vtTLS. In the worst case, one additional packet may be sent if the encryption function requires fixed block size and the maximum size of the packet, after the second MAC and the second encryption, is exceeded by, at least, one byte. In this case, an additional full packet is needed due to the constraint of having fixed block size.

5 Evaluation

We evaluated vtTLS in terms of two aspects: *performance* and *cost*. We considered OpenSSL 1.0.2g as the baseline, due to the fact that vtTLS is based on that software and version.

Diversity has performance costs and creates overhead in the communication. Every message sent needs to be ciphered and signed $k-1$ times more than using a TLS implementation and every message received needs to be deciphered and verified also $k-1$ times more. In the worst case, users should experience a connection k times slower than using OpenSSL. We considered $k = 2$ in all experiments, as this is the value we expect to be used in practice (we expect vulnerabilities to appear rarely, so the ability to tolerate one vulnerability per mechanism sufficient). With this experimental evaluation, we want to be able to state if vtTLS is a viable mechanism for daily usage, i.e., if the penalty for replacing TLS channels by vtTLS channels is not prohibitive.

In order to perform these tests, we used two virtual machines in the same Intel Core i7 computer with 8 GB RAM. The virtual machines run Debian 8 and openSUSE 12 playing the roles of server and client, respectively. All the tests were done in the same controlled environment and same geographic location.

Table 1 Handshake time comparison.

	Average (ms)	Standard deviation	Confidence interval (95%)
vtTLS	3.909	0.963	±0.180
OpenSSL	2.345	0.933	±0.174

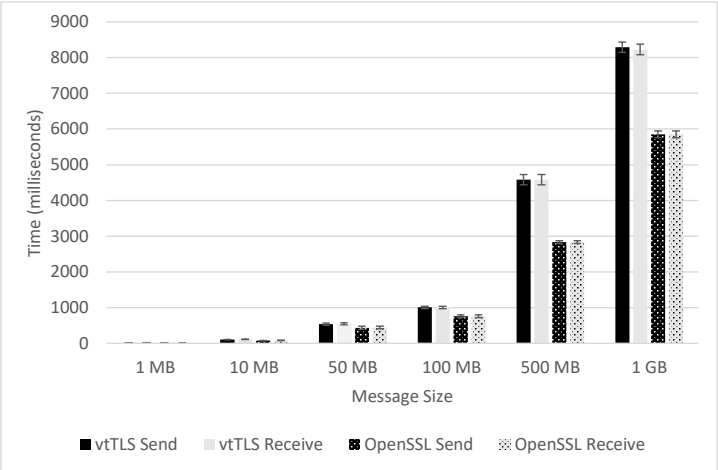


Figure 4 Time to send and receive a message with vtTLS and OpenSSL.

5.1 Performance

In order to evaluate the performance of vtTLS, we executed several tests. The main goal was to understand if the overhead of vtTLS is lower, equal, or bigger than k times in relation to OpenSSL. We configured vtTLS to use the following cipher suites: `TLS_RSA_WITH_AES_256_GCM_SHA384` and `TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384`. The suite used with OpenSSL was the latter.

To evaluate the performance of the handshake, we executed 100 times the Handshake Protocol of both vtTLS and OpenSSL. In average, the vtTLS handshake took 3.909 milliseconds to conclude and the OpenSSL handshake 2.345 milliseconds. Therefore, the vtTLS handshake is only 1.67 slower than the OpenSSL handshake, which is better than the worst case. Table 1 provides more details.

After evaluating the Handshake, we performed data communication tests to assess the overhead generated by the diversity and redundancy of mechanisms. As the Handshake, the communication is expected to be at most $k = 2$ times slower than a TLS communication. For this test, we considered a sample of 100 messages sent and received with vtTLS and 100 messages sent and received with OpenSSL.

Figure 4 shows the comparison between the time it takes to send and receive a message with vtTLS and OpenSSL/TLS. Tables 2 and 3 show more details.

The measurements concern the time each channel needs to perform the local operations in order to send the message (including encryption, signing, second encryption and second signing). These values do not include the time taken by the message to reach its destination through the network. The timer is started before the call to `SSL_write` and stopped after the function returns. As for the results regarding the reception of messages, the measured time is the time taken to perform the operations necessary to retrieve the message (including second decrypting and second verifying), i.e. the time of execution of `SSL_read`. This methodology

Table 2 Time to send a message with vtTLS and OpenSSL.

	vtTLS Send			OpenSSL Send		
	Average	**St. dev.**	**Conf. I. (95%)**	**Average**	**St. dev.**	**Conf. I. (95%)**
1 MB	12.80	3.324	±0.652	11.28	3.985	±0.781
10 MB	105.89	17.573	±3.444	76.61	10.413	±2.041
50 MB	534.55	149.697	±29.340	435.01	212.065	±41.564
100 MB	1004.30	194.701	±38.161	757.02	206.709	±40.514
500 MB	4579.40	727.519	±142.591	2834.18	217.378	±42.605
1 GB	8289.78	757.167	±148.402	5851.08	480.423	±94.161

Table 3 Time to receive a message with vtTLS and OpenSSL.

	vtTLS Receive			OpenSSL Receive		
	Average	**St. dev.**	**Conf. I. (95%)**	**Average**	**St. dev.**	**Conf. I.(95%)**
1 MB	14.70	3.324	±0.652	13.48	3.985	±0.781
10 MB	113.41	17.573	±3.444	80.42	10.413	±2.041
50 MB	549.61	149.697	±29.340	443.10	212.065	±41.564
100 MB	1004.54	194.701	±38.161	757.13	206.709	±40.514
500 MB	4580.13	727.519	±142.591	2834.37	217.378	±42.605
1 GB	8227	757.167	±148.402	5850.96	480.423	±94.161

is only possible due to the fact that `SSL_write` is a synchronous call. It only returns after writing the message to the buffer. And also due to the fact that `SSL_read` is also synchronous as it only returns when the message is read.

In average, a message sent through a vtTLS channel takes 22.88% longer than a message sent with OpenSSL. For example, a 50 MB message takes an average of 534.55 ms to be sent with vtTLS. With OpenSSL, the same message takes 435.01 ms to be sent. The overhead generated by using diverse encryption and MAC mechanisms exists, as expected, but it is much smaller than the expected worst case.

In order to validate the premise that the message increase is the same considering the same message size, we measured the increase in the message size comparing once again vtTLS and OpenSSL channels. A 100 KB plaintext message converts into a ciphertext of 102,771 bytes with vtTLS. With OpenSSL, the same message corresponds to a ciphertext of 102,603 bytes. Concluding, sending a 100 KB message through vtTLS costs an additional 168 bytes. Therefore, as stated before, the number of extra bytes sent is not directly proportional to the message size.

We also evaluated the message size of the ciphertext of a 1 MB plaintext message. A 1 MB plaintext message corresponds to a ciphertext of 1,029,054 bytes using vtTLS, while using OpenSSL the same message has 1,025,856 bytes. Concluding, sending 1 MB through a vtTLS channel costs an additional 3,198 bytes than using a OpenSSL channel. Table 4 shows all the results obtained and the comparison between the message sizes.

5.2 Cost

Similarly to TLS, vtTLS uses certificates that require some management effort and costs. A server using OpenSSL/ TLS to protect the communication with clients needs only one certificate. If the administrator decides to use vtTLS instead of TLS, at least 2 certificates are needed for maximum diversity, and at most k certificates. Although certificates are not

Table 4 Message sizes in vtTLS and OpenSSL.

Message size	vtTLS		OpenSSL		Overhead (diff.)	
	Encrypted message size	Average #packs	Encrypted message size	Average #packs	Packets	Message size
100,000	102,771	6.3	102,603	5.3	1	168
1,000,000	1,029,054	38.3	1,025,856	37.6	0.7	3,198
100,000,000	105,362,077.10	2,830.2	104,956,194.50	2,553.5	276.7	405,883

expensive, they represent a cost. vtTLS can be used with just one certificate, but this reduces the diversity and, consequently, the potential security benefit.

Regarding management, there is the need to manage two certificates instead of one. We believe this does not represent a substantial increase in management effort. If it is decided to use vtTLS with a diversity factor $k > 2$, the management costs of maintaining k certificates might represent an significant increase of management costs.

6 Conclusions

vtTLS is a diverse and redundant vulnerability-tolerant secure communication protocol designed for communication on the Internet. It aims at increasing security using diverse cipher suites to tolerate vulnerabilities in the encryption mechanisms used in the communication channel. In order to evaluate our solution, we compared it to an OpenSSL 1.0.2g communication channel. While expected to be $k = 2$ times slower than an OpenSSL channel, the evaluation showed that using diversity and redundancy of cryptographic mechanisms in vtTLS does not generate such a high overhead. vtTLS takes, in average, 22.88% longer to send a message than TLS/OpenSSL, but considering the increase in security, this overhead is acceptable. Overall, considering the additional costs of having an extra certificate, the time increase, and potential management costs, vtTLS provides an interesting trade-off for a set of critical applications.

References

1 D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. Vandersloot, E. Wustrow, and S. Paul. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 5–1, 2015.

2 M. R. Albrecht, J. P. Degabriele, T. B. Hansen, and K. G. Paterson. A surfeit of SSH cipher suites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491, 2016.

3 A. Avižienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the IEEE International Computer Software and Applications Conference*, pages 149–155, 1977.

4 L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 833–844, 2012.

5 A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full AES. In *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security*, volume LNCS 7073, pages 344–371, 2011.

6 M. Carvalho, J. DeMott, R. Ford, and D. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014.

7 M. Carvalho and R. Ford. Moving-target defenses for computer networks. *IEEE Security and Privacy*, 12(2):73–76, 2014.

8 R. Carvalho. Authentication security through diversity and redundancy for cloud computing. Master's thesis, Instituto Superior Técnico, Lisbon, Portugal, 2014.

9 T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.2 (RFC 5246), 2008.

10 ENISA. Algorithms, key size and parameters report – 2014, nov 2014.

11 D. Evans, A. Nguyen-Tuong, and J. Knight. Effectiveness of moving target defenses. In *Moving Target Defense*, volume 54, pages 29–48. Springer, 2011.

12 N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Proceedings of Fast Software Encryption*, volume LNCS 1978, pages 213–230. Springer, 2001.

13 M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, pages 383–394, 27–30 June 2011.

14 V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for SSL. In *Proceedings of the 1st ACM Workshop on Wireless Security*, pages 87–94, 2002.

15 A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–11, 2013.

16 A. Joaquim, M. L. Pardal, and M. Correia. vtTLS: A vulnerability-tolerant communication protocol. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*, pages 212–215, 2016.

17 S. Kanno and M. Kanda. Addition of the Camellia cipher suites to transport layer security (TLS) (RFC 6367), 2011.

18 S. Kent and K. Seo. Security architecture for the internet protocol (RFC 4301), 2005.

19 T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, volume LNCS 6223, pages 333–350, 2010.

20 P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

21 B. Littlewood and L. Strigini. Redundancy and diversity in security. In *Computer Security – ESORICS 2004, 9th European Symposium on Research Computer Security*, pages 227–246, 2004.

22 A. Menezes, P. van Oorschot, and S. Vanstone. Hash functions and data integrity. In *Handbook of Applied Cryptography*, chapter 9. CRC Press, 1996.

23 A. Menezes, P. van Oorschot, and S. Vanstone. Public-key encryption. In *Handbook of Applied Cryptography*, chapter 8. CRC Press, 1996.

24 R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford, CA, USA, 1979.

25 V. Rijmen and J. Daemen. Advanced encryption standard. *U.S. National Institute of Standards and Technology (NIST)*, 2009:8–12, 2001.

26 Tom Roeder and Fred B. Schneider. Proactive obfuscation. *ACM Trans. Comput. Syst.*, 28(2):4:1–4:54, 2010. doi:10.1145/1813654.1813655.

27 R. Seggelmann, M. Tuexen, and M. Williams. Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension (RFC 6520), 2012.

28 Y. Sheffer, R. Holz, and P. Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS) (RFC 7457), 2015.

29 P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Scientific and Statistical Computing*, 26:1484, 1995.

30 P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

31 M. Stevens. *Attacks on Hash Functions and Applications*. PhD thesis, Mathematical Institute, Leiden University, 2012.

32 M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. *IACR Cryptology ePrint Archive*, 2017:190, 2017.

33 M. Stevens, P. Karpman, and T. Peyrin. Freestart collision on full SHA-1. Cryptology ePrint Archive, Report 2015/967, 2015.

34 Paulo Veríssimo, Nuno Ferreira Neves, and Miguel Correia. Intrusion-tolerant architectures: Concepts and design. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, editors, *Architecting Dependable Systems [the book is a result of the ICSE 2002 Workshop on Software Architectures for Dependable Systems]*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer, 2002. doi:10.1007/3-540-45177-3_1.

35 J. Viega, M. Messier, and P. Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. O'Reilly, 2002.

36 X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology*, pages 17–36. Springer-Verlag, 2005.

37 J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, 2003.

38 T. Ylonen and C. Lonvick. The secure shell (SSH) protocol architecture (RFC 4251), 2006.

Asynchronous Message Orderings Beyond Causality

Adam Shimi[1], Aurélie Hurault[2], and Philippe Quéinnec[3]

1 **IRIT - Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France**
2 **IRIT - Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France**
3 **IRIT - Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France**

Abstract

In the asynchronous setting, distributed behavior is traditionally studied through computations, the Happened-Before posets of events generated by the system. An equivalent perspective considers the linear extensions of the generated computations: each linear extension defines a sequence of events, called an execution. Both perspective were leveraged in the study of asynchronous point-to-point message orderings over computations; yet neither allows us to interpret message orderings defined over executions. Can we nevertheless make sense of such an ordering, maybe even use it to understand asynchronicity better?

We provide a general answer by defining a topology on the set of executions which captures the fundamental assumptions of asynchronicity. This topology links each message ordering over executions with two sets of computations: its closure, the computations for which at least one linear extension satisfies the predicate; and its interior, the computations for which all linear extensions satisfy it. These sets of computations represent respectively the uncertainty brought by asynchronicity – the computations where the predicate is satisfiable – and the certainty available despite asynchronicity – the computations where the predicate must hold. The paper demonstrates the use of this topological approach by examining closures and interiors of interesting orderings over executions.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Asynchronous computations, Point-to-point message orderings, Causality, Topology, Interior, Closure

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.29

1 Introduction

1.1 Motivation

What can we know about the ordering of events in an asynchronous world? Only the causal order, answered Lamport's seminal work [20]. This insight grounds the description of asynchronous behavior through computations, the posets generated by events with their causal order – or equivalently through the linear extensions of these computations, sequences of events called executions.

It follows that message orderings defined over computations have been well studied, because they capture the type of constraints allowed by causality. One prominent example is the causal message ordering: it enforces that two messages whose send events are causally ordered and which share the same destination peer, must be received in the order they were

[1] This work was supported by project PARDI ANR-16-CE25-0006.

sent. Charron-Bost et al. [8] offer multiple equivalent characterizations of this ordering, as well as its place in a hierarchy with other orderings. Murty and Garg [22] prove that causal ordering is the most constrained ordering implementable using only messages tags – that is, without altering the causal order through control messages. Numerous works, among others Fidge [14], Mattern [21], Schwarz and Mattern [25], Raynal et al.[23] and Kshemkalyani and Singhal [19], explore the means of implementation of causal ordering.

But predicates over computations have drawbacks: they are defined on posets, objects less intuitive than sequences; and they do not capture every interesting constraint. For example, Fifo_{n-n} ordering formalized in Chevrou et al. [11] requires that all receptions follow the same order as their respective sends, even if neither pair of events is causally ordered. Fifo_{n-n}'s definition is simple, and it reduces asynchronous communication to a single FIFO queue. But since it demands a stronger order than the causal one, it conflicts with the very foundations of asynchronous distributed systems. As we will see, there is a general way to consider any predicate over executions in an asynchronous setting, which gives us both a powerful tool and a deeper understanding of asynchronicity.

1.2 Computations and Executions

▶ **Definition 1** (Events)**.** Let MES the set of messages and PEERS the set of communicating peers. Then EVENTS $\triangleq \{send, receive\} \times MES \times PEERS$, where the components correspond respectively to the type of events, the message and the peer where the event happens.

We write $peer(e)$ for the projection of an event into its peer component. Since in the following we will only consider sets of events where multiple sends and/or receptions of the same message are forbidden, events are considered uniquely characterized by their type and their message. We thus abuse notation by writing $s(m)$ and $r(m)$ instead of $\langle send, m, peer\rangle$ and $\langle receive, m, peer\rangle$ respectively. Note that we do not take into account internal events, since we are only interested in the order of communication ones.

▶ **Definition 2** (Computation)**.** Let X be a set of events containing at most one reception by message and one send event by message. Then the partially ordered set (or poset) $x = (X, \prec_c^x)$ is a **computation** iff

- $\prec_c^x$ is a causal order on X, that is the smallest partial order such that

$$\left(\begin{array}{lcll} peer(e_1) = peer(e_2) & \implies & e_1 \prec_c^x e_2 \vee e_2 \prec_c^x e_1 & \text{(Peer order)} \\ \exists m : e_1 = s(m) \wedge e_2 = r(m) & \implies & e_1 \prec_c^x e_2 & \text{(Message transfer)} \\ \exists e \in X : e_1 \prec_c^x e \wedge e \prec_c^x e_2 & \implies & e_1 \prec_c^x e_2 & \text{(Transitivity)} \end{array}\right).$$

- Any received message has been sent: $\forall m \in MES : r(m) \in x \implies s(m) \in x$.
- No message is sent to oneself: $\forall m \in MES : peer(r(m)) \neq peer(s(m))$.

The peer order $\prec_p^x$ is defined as the projection of $\prec_c^x$ on pairs of events happening on the same peer. When two events e_1 and e_2 are not causally ordered, we write $e_1 \parallel_c^x e_2$. We note $Comp$ the set of all computations over subsets of $EVENTS$. Finally, for b a predicate over computations, $Comp(b)$ is the set of computations satisfying b.

▶ **Definition 3** (Execution)**.** Let Σ be a set of events containing at most one reception by message and one send event by message. Then the totally ordered set $\sigma = (\Sigma, \prec^\sigma)$ is an **execution** iff $\exists x = (X, \prec_c^x) \in Comp$ such that $\Sigma = X$ and $\prec^\sigma$ is a linear extension of $\prec_c^x$. By the minimality of the causal order, such x is unique.

We write $comp(\sigma)$ for x, $\prec_c^\sigma$ for $\prec_c^x$ and $\prec_p^\sigma$ for $\prec_p^x$. We note $Exec$ the set of all executions over subset of $EVENTS$. Finally, for b a predicate over executions, $Exec(b)$ is the set of executions satisfying b.

Both are represented with so-called space-time diagrams, where each peer is represented by a vertical line, events at a peer are ordered from left to right and messages are drawn by connecting a send event with the corresponding reception. For example, Figure 3 defines the execution $s(m_1)r(m_1)s(m_2)r(m_2)$ as well as the computation over the same events and with partial order $\{(s(m_1), r(m_1)), (s(m_2), r(m_2)), (s(m_1), s(m_2)), ((s(m_1), r(m_2)))\}$.

1.3 Message Orderings

We conclude these preliminary definitions by introducing predicates over computations and executions. Since we consider only communication events, we will call these predicates message orderings. Table 1 states the message orderings we will study in the following. They are given as in Chevrou et al. [11] (except for RSC, where our definition is equivalent and easier to manipulate).

- Fifo_{1-1} ordering is the classical Fifo ordering between every pair of peers.
- Causal ordering is the ordering where messages to the same peer and with causally ordered send events are received according to their send order.
- Fifo_{n-1} ordering is the "mailbox" ordering that is used notably by [2], where messages to a peer are put into its mailbox, and the peer retrieves them according to their send order.
- Fifo_{1-n} ordering is the dual of Fifo_{n-1}. It can be thought of as a "sending box" by peer, in which each peer put messages it sends, and from which receivers fetch messages according to their send order.
- Fifo_{n-n} ordering is the ordering where all messages are received according to their send order, even if when neither their send events nor their receptions are causally ordered.
- RSC ordering (Realizable with Synchronous Communication) requires that every reception is immediately preceded by its corresponding send event.

The sets of executions defined by these models form a hierarchy, as stated and proved in [11].

▶ **Lemma 4** (Ordering hierarchy over executions).

1. $Exec(\text{RSC}) \subsetneq Exec(\text{Fifo}_{n-n}) \subsetneq Exec(\text{Fifo}_{1-n}) \subsetneq Exec(\text{Causal}) \subsetneq Exec(\text{Fifo}_{1-1})$
2. $Exec(\text{RSC}) \subsetneq Exec(\text{Fifo}_{n-n}) \subsetneq Exec(\text{Fifo}_{n-1}) \subsetneq Exec(\text{Causal}) \subsetneq Exec(\text{Fifo}_{1-1})$
3. $Exec(\text{Fifo}_{n-1}) \not\subset Exec(\text{Fifo}_{1-n})$ *and* $Exec(\text{Fifo}_{1-n}) \not\subset Exec(\text{Fifo}_{n-1})$

Turning to computations, only Fifo_{1-1} and Causal are well-defined over them, since they are defined only in terms of causal and peer order. Those two orderings obey the same hierarchy as their execution-based counterparts, as shown in Charron-Bost et al. [8].

▶ **Lemma 5.** $Comp(\text{Causal}) \subsetneq Comp(\text{Fifo}_{1-1})$

1.4 Overview of the Results

Our results are threefold.

- First, we leverage elementary topology to link any message ordering over executions with two sets of computations: its closure, corresponding to existential quantification over linear extensions of the ordering; and its interior, corresponding to universal quantification. Both sets provide a mean to study the message ordering in an asynchronous context, where computations are the least distinguishable unit.

Table 1 Ordering predicates over executions.

Name	Expression
Fifo_{1-1}	$\forall m_1, m_2 \in MES : \left(\begin{array}{ll} & r(m_1), r(m_2) \in \sigma \\ \wedge & peer(r(m_1)) = peer(r(m_2)) \\ \wedge & s(m_1) \prec_p^\sigma s(m_2) \end{array} \right) \implies r(m_1) \prec_p^\sigma r(m_2)$
Causal	$\forall m_1, m_2 \in MES : \left(\begin{array}{ll} & r(m_1), r(m_2) \in \sigma \\ \wedge & peer(r(m_1)) = peer(r(m_2)) \\ \wedge & s(m_1) \prec_c^\sigma s(m_2) \end{array} \right) \implies r(m_1) \prec_p^\sigma r(m_2)$
Fifo_{n-1}	$\forall m_1, m_2 \in MES : \left(\begin{array}{ll} & r(m_1), r(m_2) \in \sigma \\ \wedge & peer(r(m_1)) = peer(r(m_2)) \\ \wedge & s(m_1) \prec^\sigma s(m_2) \end{array} \right) \implies r(m_1) \prec_p^\sigma r(m_2)$
Fifo_{1-n}	$\forall m_1, m_2 \in MES : \left(\begin{array}{ll} & r(m_1), r(m_2) \in \sigma \\ \wedge & s(m_1) \prec_p^\sigma s(m_2) \end{array} \right) \implies r(m_1) \prec^\sigma r(m_2)$
Fifo_{n-n}	$\forall m_1, m_2 \in MES : \left(\begin{array}{ll} & r(m_1), r(m_2) \in \sigma \\ \wedge & s(m_1) \prec^\sigma s(m_2) \end{array} \right) \implies r(m_1) \prec^\sigma r(m_2)$
RSC	$\forall m_1, m_2 \in MES : \left(\begin{array}{ll} & r(m_1) \in \sigma \\ \wedge & s(m_1) \prec^\sigma s(m_2) \end{array} \right) \implies r(m_1) \prec^\sigma s(m_2)$

- Second, we characterize the closures of our message orderings (see Figure 1) by forbidden patterns in the causal order. This in turn yields a precise understanding of whether they can be distinguished at the computation level, as well as their comparative discriminating power.
- Third, we turn to interiors of the aforementioned orderings, and characterize them through the concept of chain (see Figure 2). Such characterizations expand our understanding of the orderings implementation: they provide system-level guidelines that allows us to circumvent the result of [22] for ensuring a message ordering.

The rest of the paper is organized as follows. Section 2 develops the topological approach to message orderings over executions. Then the closures and interiors of the message orderings mentioned above are studied respectively in Sections 3 and 4. Finally, Section 5 surveys related works while conclusions and perspectives are drawn in Section 6.

2 A Topological Bridge

Recall that a message ordering over executions characterizes a set of executions. On the other hand, we know from [20] that the only meaningful sets of executions in an asynchronous world correspond to the sets of all the linear extensions of a given computation, or the union of such sets. The subtlety here stems from the "all": a truncated set of linear extensions asks for an order with more discriminating power than the causal one. Making sense of any set of executions in terms of computations – and by extension in terms of causal order – will allow us to interpret any message ordering in the asynchronous setting.

First, we need a formal definition of what we call meaningful sets of executions, that is the sets defined by all the linear extensions of a given computation. They can be characterized as the equivalence classes from the causal equivalence relation.

▶ **Definition 6** (Causal Equivalence). Let $\sigma, \sigma' \in Exec$. Then $\sigma \equiv_c \sigma' \triangleq comp(\sigma) = comp(\sigma')$. We say σ and σ' are causally equivalent and write $[\sigma]_{\equiv_c}$ for the equivalence class of σ.

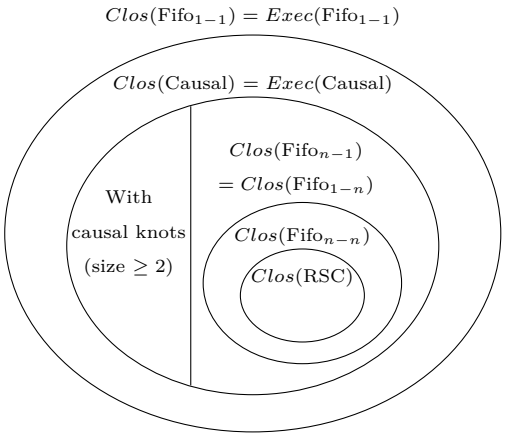


Figure 1 The closures of our message orderings.

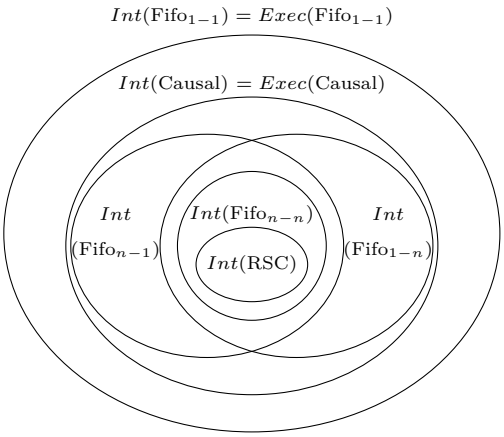


Figure 2 The interiors of our message orderings.

The quotient set of $Exec$ by causal equivalence is thus isomorphic with $Comp$. The causal equivalence classes are only the building block: any union of such blocks is also a meaningful set for our endeavors. These properties correspond to the topological concept of an open set, and motivate our introduction of the following simple topology over $Exec$.

▶ **Definition 7** (Open Set). Let $S \subseteq Exec$. Then S is an open set in the **Computation Topology** iff $\exists X \in \mathcal{P}(Exec)$ such that $S = \bigcup_{x \in X} [x]_{\equiv_c}$, where $\mathcal{P}(Exec)$ is the powerset of $Exec$.

A property of our topology that will prove useful is that any open set is also a closed one. Indeed, by definition of an equivalence relation, any union of equivalence classes is the complement of a union of equivalence classes; thus every open set is also closed. This might strike the reader used to more classical topologies as odd. Yet it allows us to state in terms of open sets both the closure and interior of a set, the two basic operations in elementary topology.

These extensions can be interpreted respectively as existential and universal quantification over equivalence classes: the set of equivalence classes (or equivalently, of computations) containing at least one execution from the original set; and the set of equivalence classes containing only executions from the original set. Intuitively, this translates respectively into the set of computations where a message ordering might hold, and the set of computations where it necessarily holds.

▶ **Definition 8** (Interior and Closure). Let $S \subseteq Exec$.

$Clos(S)$, the closure of S, is the smallest closed set such that $S \subseteq Clos(S)$.

$Int(S)$, the interior of S, is the greatest open set such that $Int(S) \subseteq S$.

We write $Clos(b)$ and $Int(b)$ for the closure and interior of the set of executions defined by the predicate b.

▶ **Lemma 9** (Collapse Lemma). *Let $S \subseteq Exec$. Then $S = Clos(S) \iff S = Int(S)$.*

Proof. ($\Rightarrow$) Since $S = Clos(S)$, S is a closed set. Given our topology, it is thus also an open set. It is therefore the greatest open set contained by itself.

($\Leftarrow$) Since $S = Int(S)$, S is an open set. Given our topology, it is thus also a closed set. It is therefore the smallest closed set containing itself. ◀

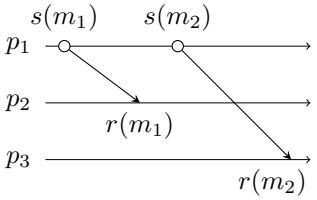


Figure 3 Fifo_{1-n} does not collapse.

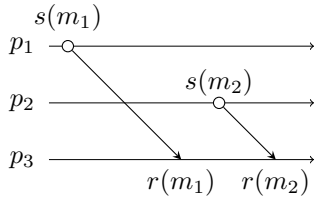


Figure 4 Fifo_{n-1} does not collapse.

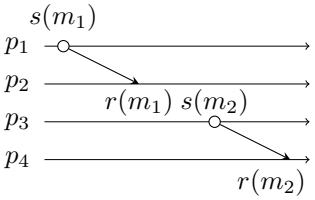


Figure 5 Fifo_{n-n} and RSC do not collapse.

We now show that the collapse described by Lemma 9 happens for Fifo_{1-1} and Causal but not the other orderings. Our topology thus captures the distinction between message orderings on executions which translate straightforwardly over computations, and message orderings on executions "asking" for more than causal order.

▶ **Theorem 10** (Fifo_{1-1} and Causal).
1. $Exec(\text{Fifo}_{1-1}) = Int(\text{Fifo}_{1-1}) = Clos(\text{Fifo}_{1-1})$
2. $Exec(\text{Causal}) = Int(\text{Causal}) = Clos(\text{Causal})$

Proof.
1. Fifo_{1-1} is defined only in terms of peer order, which is a projection of the causal order. It is therefore invariant by causal equivalence. We conclude that $Exec(\text{Fifo}_{1-1}) = Clos(\text{Fifo}_{1-1})$, which by Lemma 9 entails $Exec(\text{Fifo}_{1-1}) = Int(\text{Fifo}_{1-1}) = Clos(\text{Fifo}_{1-1})$.
2. Causal is invariant by causal equivalence for the same reason as Fifo_{1-1}. We conclude that $Exec(\text{Causal}) = Clos(\text{Causal})$, which by Lemma 9 gives us $Exec(\text{Causal}) = Int(\text{Causal}) = Clos(\text{Causal})$. ◀

▶ **Theorem 11** (Fifo_{n-1}, Fifo_{1-n}, Fifo_{n-n} and RSC).
1. $Int(\text{Fifo}_{1-n}) \subsetneq Exec(\text{Fifo}_{1-n}) \subsetneq Clos(\text{Fifo}_{1-n})$
2. $Int(\text{Fifo}_{n-1}) \subsetneq Exec(\text{Fifo}_{n-1}) \subsetneq Clos(\text{Fifo}_{n-1})$
3. $Int(\text{Fifo}_{n-n}) \subsetneq Exec(\text{Fifo}_{n-n}) \subsetneq Clos(\text{Fifo}_{n-n})$
4. $Int(\text{RSC}) \subsetneq Exec(\text{RSC}) \subsetneq Clos(\text{RSC})$

Proof. We only consider the cases where $EVENTS$ is non trivial, which here means that it contains at least two send events and two receptions.

The inclusions then follow directly from the definition of Int and $Clos$. As for strictness, counter-examples separating either the interior or the closure with the initial sets are enough: Lemma 9 then yields the strictness of inclusion for both.
1. Figure 3 is in $Exec(\text{Fifo}_{1-n})$ but not in its interior: the execution $s(m_1)s(m_2)r(m_2)r(m_1)$ is causally equivalent to it while breaking the Fifo_{1-n} predicate. Thus $Exec(\text{Fifo}_{1-n}) \neq Int(\text{Fifo}_{1-n})$.
2. Figure 4 is in $Exec(\text{Fifo}_{n-1})$ but not in its interior: the execution $s(m_2)s(m_1)r(m_1)r(m_2)$ is causally equivalent to it while breaking the Fifo_{n-1} predicate. Thus $Exec(\text{Fifo}_{n-1}) \neq Int(\text{Fifo}_{n-1})$.
3. Figure 5 is in $Exec(\text{Fifo}_{1-n})$ but not in its interior: the execution $s(m_1)s(m_2)r(m_2)r(m_1)$ is causally equivalent to it, while breaking the Fifo_{n-n} predicate. Thus $Exec(\text{Fifo}_{n-n}) \neq Int(\text{Fifo}_{n-n})$.
4. Figure 5 is also in $Exec(\text{RSC})$ and not in its interior: the execution exhibited in the previous case breaks RSC too. Thus $Exec(\text{RSC}) \neq Int((\text{RSC}))$. ◀

3 Closures

3.1 Characterization of the Closure of RSC

When we apply the closure operation on our RSC over executions, we obtain the set of computations with at least one RSC linear extension. This in turn corresponds to the RSC over computations from Charron-Bost et al. [8]. We thus reuse the characterization of $Clos$(RSC) by Charron-Bost et al., through forbidden patterns in the causal order named crowns.

▶ **Definition 12** (Crown). Let σ an execution. Then a set of messages $m_1, .., m_n$ forms a **crown** of size $n \iff \forall i \in [1, n] : s(m_i) \prec_c^\sigma r(m_{i+1})$, where $m_{n+1} = m_1$.

For instance, Figure 6 is a crown of size 2.

▶ **Theorem 13** (Crown Characterization). *Let $\sigma \in Exec$. Then $\sigma \in Clos$(RSC) $\iff$ σ contains no crown.*

Proof. See [8]. ◀

3.2 Characterization of $Clos$(Fifo$_{n-1}$) and $Clos$(Fifo$_{1-n}$)

A similar negative characterization of $Clos$(Fifo$_{n-1}$) and $Clos$(Fifo$_{1-n}$) is given by special crowns we call causal knots.

▶ **Definition 14** (Causal knot). Let σ an execution. Then a set of messages $m_1, .., m_{2s}$ forms a **causal knot** of size $s \iff \forall i \in [1, 2s] : \left(\begin{array}{l} i \equiv 1 \bmod 2 : s(m_i) \prec_c^\sigma s(m_{i+1}) \\ i \equiv 0 \bmod 2 : r(m_i) \prec_c^\sigma r(m_{i+1}) \end{array} \right)$,
where $m_{2s+1} = m_1$.

Figure 7 is an example of a causal knot of size 2. We show in the rest of this section that causal knots characterize $Clos$(Fifo$_{n-1}$) and $Clos$(Fifo$_{1-n}$). Let us start with $Clos$(Fifo$_{n-1}$): we first define a relation for each causal equivalence class characterizing the additional constraints on executions of this class to satisfy Fifo$_{n-1}$.

▶ **Definition 15** (N-1 order). Let $\sigma \in Exec$ and Σ the underlying set of events. Then $\lhd_{n-1}^\sigma \triangleq \{(s(m_1), s(m_2)) \in \Sigma \times \Sigma \mid r(m_1) \prec_p^\sigma r(m_2)\}$. And $\prec_{n-1}^\sigma \triangleq \prec_c^\sigma \cup \lhd_{n-1}^\sigma$.

Then the next Lemma reduces the membership of an execution σ in $Clos$(Fifo$_{n-1}$) to whether or not $\prec_{n-1}^\sigma$ contains a cycle.

▶ **Lemma 16.** *Let $\sigma \in Exec$. Then $\sigma \in Clos(Fifo_{n-1}) \iff \prec_{n-1}^\sigma$ is antisymmetric.*

Proof. ($\Rightarrow$) We show the contrapositive: If $\prec_{n-1}^\sigma$ contains a cycle, then $\sigma \notin Clos$(Fifo$_{n-1}$).

Let $\sigma \in Exec$ such that $\prec_{n-1}^\sigma$ contains a cycle. Then its transitive closure is not a partial order: it has no linear extensions. From that fact, we now prove that no execution in $[\sigma]_{\equiv_c}$ satisfies Fifo$_{n-1}$, and thus that $\sigma \notin Clos$(Fifo$_{n-1}$).

Let $\sigma' \equiv_c \sigma$. By our hypothesis that $\prec_{n-1}^\sigma$ contains a cycle, we have $\prec_{n-1}^\sigma \not\subset \prec^{\sigma'}$. But since $\sigma' \equiv_c \sigma$, we have $\prec_c^\sigma \subset \prec^{\sigma'}$. We thus conclude that $\lhd_{n-1}^\sigma \not\subset \prec^{\sigma'}$. Thus $\exists m_1, m_2$ such that $r(m_1) \prec_p^\sigma r(m_2) \wedge s(m_1) \not\prec^{\sigma'} s(m_2)$. By totality of $\prec^{\sigma'}$, we have $r(m_1) \prec_p^\sigma r(m_2) \wedge s(m_2) \prec^{\sigma'} s(m_1)$, and thus σ' violates Fifo$_{n-1}$.

($\Leftarrow$) Let $\sigma \in Exec$ such that $\prec_{n-1}^\sigma$ is antisymmetric. Then the transitive closure of $\prec_{n-1}^\sigma$ is reflexive (because $\prec_c^\sigma$ is reflexive), transitive and antisymmetric. It is therefore a partial order, who has a linear extension agreeing both with $\prec_c^\sigma$ and $\lhd_{n-1}^\sigma$. This linear extension defines an execution $\sigma' \equiv_c \sigma$ (because $\prec_c^\sigma \subset \prec^{\sigma'}$) such that σ' satisfies Fifo$_{n-1}$ (because $\lhd_{n-1}^\sigma \subset \prec^{\sigma'}$). We conclude that $\sigma \in Clos$(Fifo$_{n-1}$). ◀

▶ **Theorem 17** (Causal Knot Criterion for Fifo_{n-1}). *Let $\sigma \in Exec$. Then $\sigma \in Clos(\text{Fifo}_{n-1})$ $\iff$ σ contains no causal knot.*

Proof. By Lemma 16 and the transitivity of equivalence, we only need to prove that $\prec^\sigma_{n-1}$ is antisymmetric $\iff$ σ contains no causal knot.

($\Rightarrow$) We prove the contrapositive: if σ contains a causal knot, then $\prec^\sigma_{n-1}$ is not antisymmetric. Let $\sigma \in Exec$ with a causal knot of size k. By definition of causal knots, there are $2k$ messages $m_1, .., m_{2k}$ such that $\iff \forall i \in [1, 2k] : \left(\begin{array}{l} i \equiv 1 \bmod 2 : s(m_i) \prec^\sigma_c s(m_{i+1}) \\ i \equiv 0 \bmod 2 : r(m_i) \prec^\sigma_c r(m_{i+1}) \end{array} \right)$, where $m_{2k+1} = m_1$. We define another sequence of messages $m'_1, .., m'_{2k}$ from the messages of the causal knot: $\forall i \in [1, 2k]$:

$$\left(\begin{array}{l} i \equiv 1 \bmod 2 : m'_i = m_i \\ i \equiv 0 \bmod 2 : if\ r(m_i) \prec^\sigma_p r(m_{i+1}) \\ \qquad\qquad\ then\ m'_i = m_i \\ \qquad\qquad\ else\ m'_i = m, where\ m \in MES : r(m_i) \prec^\sigma_c s(m) \prec^\sigma_c r(m) \prec^\sigma_p r(m_{i+1}) \end{array} \right),$$

where $m'_{2k+1} = m'_1$. The case $i \equiv 0 \bmod 2$ is well-defined since the causal order corresponds to either the peer order or a chain of messages. We then have $\forall i \in [1, 2k]$: $\left(\begin{array}{l} i \equiv 1 \bmod 2 : s(m'_i) \prec^\sigma_c s(m'_{i+1}) \\ i \equiv 0 \bmod 2 : s(m'_i) \lhd^\sigma_{n-1} s(m'_{i+1}) \end{array} \right)$, where $m'_{2k+1} = m'_1$. Thus $\prec^\sigma_{n-1}$ contains a cycle and is not antisymmetric.

($\Leftarrow$) We prove the contrapositive: if $\prec^\sigma_{n-1}$ contains a cycle, then σ contains a causal knot.

Let $\sigma \in Exec$ such that $\prec^\sigma_{n-1}$ contains a cycle. Since both $\prec^\sigma_c$ and $\lhd^\sigma_{n-1}$ are partial orders, they are both antisymmetric: a minimal cycle thus consists of an alternation of those two. Considering such a minimal cycle of size s, we can deduce by transitivity of $\prec^\sigma_c$ and $\lhd^\sigma_{n-1}$ that it is composed of s events $e_1, .., e_s$ such that $\forall i \in [1, s] : \left(\begin{array}{l} i \equiv 1 \bmod 2 : e_i \prec^\sigma_c e_{i+1} \\ i \equiv 0 \bmod 2 : e_i \lhd^\sigma_{n-1} e_{i+1} \end{array} \right)$, where $e_{s+1} = e_1$. By minimality of the cycle and $e_1 \prec^\sigma_c e_2$, we have $e_s \lhd^\sigma_{n-1} e_1$ and thus $s = 0 \bmod 2$. Let $k = s/2$.

Recall that $\lhd^\sigma_{n-1}$ only orders send events: $e_1, .., e_s$ are thus send events. Their messages are distinct, by minimality of the cycle. Then the previous characterization can be rewritten as $\exists m_1, .., m_{2k} \in MES$ such that $\forall i \in [1, 2k] : \left(\begin{array}{l} i \equiv 1 \bmod 2 : s(m_i) \prec^\sigma_c s(m_{i+1}) \\ i \equiv 0 \bmod 2 : s(m_i) \lhd^\sigma_{n-1} s(m_{i+1}) \end{array} \right)$, where $m_{s+1} = m_1$. By definition of $\lhd^\sigma_{n-1}$, this is a causal knot of size k. ◀

$Clos(\text{Fifo}_{1-n})$ is also negatively characterized by causal knots. Thus $Clos(\text{Fifo}_{n-1}) = Clos(\text{Fifo}_{1-n})$.

▶ **Theorem 18** (Causal Knot Criterion for Fifo_{1-n}). *Let $\sigma \in Exec$. Then $\sigma \in Clos(\text{Fifo}_{1-n})$ $\iff$ σ contains no causal knot.*

Proof. The proof follows the exact same lines as the Causal knot criterion for Fifo_{n-1}, except that we substitute $\lhd^\sigma_{1-n} = \{(r(m_1), r(m_2)) \in \Sigma \times \Sigma \mid s(m_1) \prec^\sigma_p s(m_2)\}$ for $\lhd^\sigma_{n-1}$. It was moved to Appendix A due to space constraints. ◀

This surprising result shows that at the level of computations, those two message orderings cannot be separated.

3.3 Inclusion of closures

We lack a characterization of $Clos(\text{Fifo}_{n-n})$ in terms of forbidden patterns akin to crowns or causal knots. We can nonetheless separate it from $Clos(\text{Fifo}_{n-1}) = Clos(\text{Fifo}_{1-n})$, and from $Clos(\text{RSC})$.

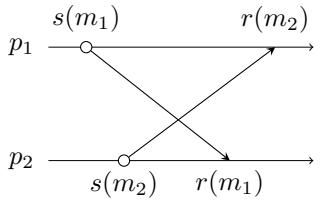


Figure 6 A crown of size 2.

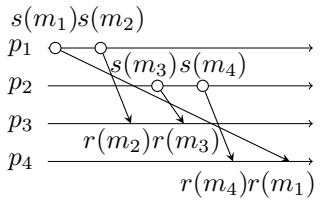


Figure 7 A causal knot of size 2.

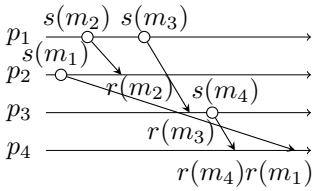


Figure 8 Not in $Clos(\text{Fifo}_{n-n})$; in $Clos(\text{Fifo}_{n-1}) = Clos(\text{Fifo}_{1-n})$.

▶ **Theorem 19** (Inclusion of closures). $Clos(\text{RSC}) \subsetneq Clos(\text{Fifo}_{n-n}) \subsetneq Clos(\text{Fifo}_{1-n}) = Clos(\text{Fifo}_{n-1})$

Proof. The inclusions follow straight from the hierarchy over executions and the definition of an open set: for $A, B \in Exec$, $A \subsetneq B$ implies that $Clos(A) \subseteq Clos(B)$. The first strictness follows from the fact that Figure 6 defines an execution in $Clos(\text{Fifo}_{n-n})$ with a crown, thus not in $Clos(\text{RSC})$. The second strictness follows from the fact that Figure 8 defines an execution σ in $Exec(\text{Fifo}_{1-n})$ (and thus in $Clos(\text{Fifo}_{1-n})$), yet this execution is not in $Clos(\text{Fifo}_{n-n})$.

Assume the contrary: $\exists \sigma' \equiv_c \sigma$ such that $\text{Fifo}_{n-n}(\sigma')$. Then $s(m_4) \prec^{\sigma'} s(m_1) \wedge r(m_2) \prec^{\sigma'} r(m_3)$, since $r(m_4) \prec_p^{\sigma} s(m_1) \wedge s(m_2) \prec_p^{\sigma} s(m_3)$. By $s(m_1) \prec_p^{\sigma} r(m_2) \wedge r(m_3) \prec_p^{\sigma} s(m_4)$, this yields $r(m_2) \prec^{\sigma'} r(m_3) \prec_p^{\sigma} s(m_4) \prec^{\sigma'} s(m_1) \prec_p r(m_2)$. **Contradiction**. ◀

3.4 Interpretation of closures

Closures capture a fundamental part of asynchronicity: the loss of knowledge it entails. Expanding the exact set of executions characterized by a message ordering to its closure introduces the additional uncertainty that comes with the non-determinism of asynchronicity.

First, closures yield a general method to check the distinguishability of a distributed system with message ordering A from one with message ordering B. For example, we deduce from the causal knot criterion that Fifo_{n-1} and Fifo_{1-n} are indistinguishable in the asynchronous world. There is no computation one can exhibit to separate them, showing with certainty which one is implemented.

Second, let us consider the complement of a closure. It represents certainty amidst asynchronicity: assurance of the non-satisfiability of the predicate. This yields a technique for ensuring that forbidden executions do not arise in a distributed system. Take the set of computations with a causal knot of size ≥ 2 as an example. These are "just causal": despite the inherent uncertainty stemming from asynchronicity, we know for sure that a system generating any of these computations cannot have a Fifo_{n-1} or Fifo_{1-n} (let alone Fifo_{n-n} or RSC) execution. Since algorithms and implementations alike usually work better with more constrained message orderings, the less constrained a computation, the more susceptible it is to cause an error or a failure. Just causal computations are thus good candidates for testing and debugging.

4 Interiors

Interiors were defined above as universal quantification over linear extensions: they thus characterize computations where, despite the inherent uncertainty of asynchronous communication, the ordering in question necessarily holds. Through chains, a simple concept from order theory, we characterize and interpret our orderings' interiors.

▶ **Definition 20** (Chain). Let $(A, \prec^A)$ a poset. Then $S \subseteq A$ is a **chain** if its elements are totally ordered for $\prec^A$: $\forall a, b \in S, a \prec^A b \vee b \prec^A a$.

We write that an execution is a chain when its set of events is a chain for its causal order.

4.1 The Interior of RSC

The interior of RSC is characterized by executions which are necessarily a chain with regard to causality. This means that all events of an execution in Int(RSC) are fully ordered by causality. Before stating the theorem, we introduce a result from dimension theory, and its corollary for inverting events in an execution while staying in the causal equivalence class.

▶ **Lemma 21** (Interpolation). *Let $(A, \prec^A)$ a poset, $S \subset A$ and $\prec_l$ a linear extension of $\prec^A_{|S}$ (where $\prec^A_{|S}$ is the projection of $\prec^A$ on $S \times S$). Then $\exists \prec_{l'}$ a linear extension of $\prec^A$ such that $\prec_{l'|S} = \prec_l$.*

Proof. It is a classical result from dimension theory. See Trotter [28] Chapter 1. ◀

▶ **Corollary 22** (Interpolation of executions). *Let σ an execution, Σ the underlying set of events, $S \subset \Sigma$ and $\prec_l$ a linear extension of $\prec^\sigma_{c|S}$. Then $\exists \sigma' \equiv_c \sigma$ such that $\prec^{\sigma'}_{|S} = \prec_l$.*

Proof. It follows immediately from Lemma 21 and the fact that a linear extension of a causal order defines an execution. ◀

▶ **Theorem 23** (Characterization of Int(RSC)). *Let σ an execution with at least one reception. Then $\sigma \in Int(\text{RSC}) \iff \sigma \in Exec(Causal) \wedge \sigma$ is a chain.*

Proof. ($\Rightarrow$) Let $\sigma \in Int$(RSC) with at least one reception. By the hierarchy of message orderings and the definition of interiors, $\sigma \in Exec$(Causal); we show that σ is a chain.

Let $s(m_1), s(m_2) \in \sigma$ such that $s(m_1) \prec^\sigma s(m_2)$. By considering all the possible cases concerning their receptions, we show that all events in σ form a chain.

1. $r(m_1) \in \sigma$. We prove by contradiction that $r(m_1) \prec^\sigma_c s(m_2)$ and thus that $s(m_1) \prec^\sigma_c r(m_1) \prec^\sigma_c s(m_2)$.
   Assume the contrary: $r(m_1) \not\prec^\sigma_c s(m_2)$. Then $\prec_l$ defined by $s(m_1) \prec_l s(m_2) \prec_l r(m_1)$ is a linear extension of $\prec^\sigma_{c|\{s(m_1), s(m_2), r(m_1)\}}$. Thus Corollary 22 ensures the existence of $\sigma' \equiv_c \sigma$ such that $s(m_1) \prec^{\sigma'} s(m_2) \prec^{\sigma'} r(m_1)$. Since σ' violates RSC, we conclude that $\sigma \notin Int$(RSC). **Contradiction**.
   We conclude that $s(m_1) \prec^\sigma_c r(m_1) \prec^\sigma_c s(m_2)$.
2. $r(m_2) \in \sigma$. Then the same reasoning by contradiction than above, here assuming $s(m_1) \not\prec^\sigma_c s(m_2)$, yields $s(m_1) \prec^\sigma_c s(m_2)$ and thus $s(m_1) \prec^\sigma_c s(m_2) \prec^\sigma_c r(m_2)$.
3. $r(m_1) \notin \sigma \wedge r(m_2) \notin \sigma$. Yet by hypothesis, σ contains at least one reception. We split by cases depending on where this reception is placed according to $s(m_1)$ and $s(m_2)$.
   - $\exists m \in MES : s(m_1) \prec^\sigma r(m) \prec^\sigma s(m_2)$. Since $\sigma \in$ RSC, we have $s(m_1) \prec^\sigma s(m) \prec^\sigma r(m) \prec^\sigma s(m_2)$. By case 1 above we have $s(m_1) \prec^\sigma_c r(m)$, and by case 2 above we have $r(m) \prec^\sigma_c s(m_2)$. Transitivity then yields $s(m_1) \prec^\sigma_c s(m_2)$.
   - $\{m \in MES \mid s(m_1) \prec^\sigma r(m) \prec^\sigma s(m_2)\} = \emptyset \wedge \exists m \in MES : r(m) \prec^\sigma s(m_1) \prec^\sigma s(m_2)$. We take m such that it is the last received message before both send events. Then by the case 1 above, $r(m) \prec^\sigma_c s(m_1) \wedge r(m) \prec^\sigma_c s(m_2)$. By definition of m, this gives us $r(m) \prec^\sigma_p s(m_1) \wedge r(m) \prec^\sigma_p s(m_2)$.
     We conclude $s(m_1) \prec^\sigma_p s(m_2)$ and thus $s(m_1) \prec^\sigma_c s(m_2)$.

- $\{m \in MES \mid s(m_1) \prec^\sigma r(m) \prec^\sigma s(m_2)\} = \emptyset \wedge \exists m \in MES : s(m_1) \prec^\sigma s(m_2) \prec^\sigma r(m)$. We take m such that it is the first received message after both send events. The same reasoning as in the previous case, using case 2 instead of case 1, yields $s(m_1) \prec_p^\sigma s(m_2)$ and thus $s(m_1) \prec_c^\sigma s(m_2)$.

($\Leftarrow$) We prove the contrapositive: $\sigma \notin Int(\text{RSC}) \implies \sigma \notin Exec(\text{Causal}) \vee \sigma$ not a chain.

Let σ an execution with at least one reception such that $\sigma \notin Int(\text{RSC})$. Thus $\exists \sigma' \equiv_c \sigma$ such that $\exists m_1, m_2 \in MES : s(m_1) \prec^{\sigma'} s(m_2) \prec^{\sigma'} r(m_1)$. We then have two possible cases.

- $s(m_1) \prec_c^\sigma s(m_2) \prec_c^\sigma r(m_1)$. Then $\exists m \in MES$ such that either $s(m_1) \prec_c^\sigma s(m_2) \prec_c^\sigma s(m) \prec_c^\sigma r(m) \prec_p^\sigma r(m_1)$ or $s(m_1) \prec_c^\sigma s(m) \prec_c^\sigma r(m) \prec_p^\sigma s(m_2) \prec_p^\sigma r(m_1)$, since the causal order corresponds to either the peer order or a chain of messages and since $peer(s(m_1)) \neq peer(r(m_1))$ by definition of a computation. From $s(m_1) \prec_c^\sigma s(m) \prec_c^\sigma r(m) \prec_p^\sigma r(m_1)$, we conclude $\sigma \notin Exec(\text{Causal})$.
- $s(m_1) \not\prec_c^\sigma s(m_2) \vee s(m_2) \not\prec_c^\sigma r(m_1)$. Since $s(m_1) \prec^{\sigma'} s(m_2) \prec^{\sigma'} r(m_1)$, we have $s(m_2) \not\prec_c^\sigma s(m_1) \wedge r(m_1) \not\prec_c^\sigma s(m_2)$. Thus σ is not a chain. ◀

All executions in $Int(\text{RSC})$ are therefore totally determined by their causal order (except the degenerate ones without receptions). This is an extraordinarily strong condition, since it means the causal equivalence class of the execution is the singleton of the execution itself!

4.2 The Interior of Fifo$_{n-n}$

By relaxing the previous characterization, we derive a characterization for $Int(\text{Fifo}_{n-n})$ based on two chains: one for the send events of received messages, and one for receptions. This results in two differences with $Int(\text{RSC})$. First, send events of never received messages are not constrained. Second, consecutive send events are possible even when the corresponding messages are eventually received. In this case, the send events must happen on the same peer; the corresponding receptions must also happen on the same peer.

▶ **Theorem 24** (Characterization of $Int(\text{Fifo}_{n-n})$). *Let $\sigma \in Exec$, $r_\sigma = \{r(m) \in \sigma\}$, and $s_\sigma = \{s(m) \in \sigma \mid r(m) \in \sigma\}$. Then $\sigma \in Int(\text{Fifo}_{n-n}) \iff \sigma \in Exec(Causal) \wedge r_\sigma$ is a chain $\wedge$ s_σ is a chain.*

Proof. This proof and the next two follow the same scheme as the proof of theorem 23 (see Appendix B). ◀

4.3 The Interiors of Fifo$_{n-1}$ and Fifo$_{1-n}$

The characterization of $Int(\text{Fifo}_{n-n})$ is itself weakened in two distinct ways to yield those of $Int(\text{Fifo}_{n-1})$ and $Int(\text{Fifo}_{1-n})$: in $Int(\text{Fifo}_{n-1})$, only send events to a same peer are required to form chains, while in $Int(\text{Fifo}_{1-n})$ it is the receptions from the same peer that need to form chains.

▶ **Theorem 25** (Characterization of $Int(\text{Fifo}_{n-1})$). *Let $\sigma \in Exec$, and $s_\sigma(p) = \{s(m) \in \sigma \mid r(m) \in \sigma \wedge peer(r(m)) = p\}$. Then $\sigma \in Int(\text{Fifo}_{n-1}) \iff \sigma \in Exec(Causal) \wedge \forall p \in PEERS, s_\sigma(p)$ is a chain.*

▶ **Theorem 26** (Characterization of $Int(\text{Fifo}_{1-n})$). *Let $\sigma \in Exec$, and $r_\sigma(p) = \{r(m) \in \sigma \mid peer(s(m)) = p\}$. Then $\sigma \in Int(\text{Fifo}_{1-n}) \iff \sigma \in Exec(Causal) \wedge \forall p \in PEERS, r_\sigma(p)$ is a chain.*

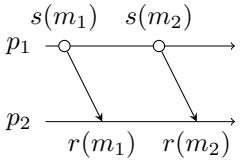


Figure 9 In $Int(\text{RSC})$; not in $Int(\text{Fifo}_{n-n})$

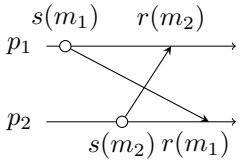


Figure 10 In $Int(\text{Fifo}_{n-n})$; not in $Int(\text{Fifo}_{1-n})$ or $Int(\text{Fifo}_{n-1})$

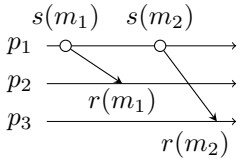


Figure 11 In $Int(\text{Fifo}_{n-1})$; not in $Int(\text{Fifo}_{1-n})$

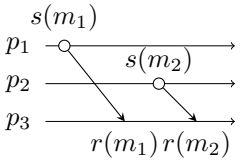


Figure 12 In $Int(\text{Fifo}_{1-n})$; not in $Int(\text{Fifo}_{n-1})$

4.4 Inclusion of FIFO Interiors

We close this study of interiors by showing the various inclusions between them. These turn out to form the same hierarchy as the initial orderings.

▶ **Theorem 27.**

1. $Int(\text{RSC}) \subsetneq Int(\text{Fifo}_{n-n}) \subsetneq Int(\text{Fifo}_{1-n})$
2. $Int(\text{RSC}) \subsetneq Int(\text{Fifo}_{n-n}) \subsetneq Int(\text{Fifo}_{n-1})$
3. $Int(\text{Fifo}_{n-1}) \not\subset Int(\text{Fifo}_{1-n})$ *and* $Int(\text{Fifo}_{1-n}) \not\subset Int(\text{Fifo}_{n-1})$

Proof. The inclusions follow straight from the hierarchy over executions and the definition of an open set: for $A, B \in Exec$, $A \subsetneq B$ implies that $Int(A) \subseteq Int(B)$. For the strictness, it is enough to give examples separating the characterizations.

1. Figure 9 separates $Int(\text{RSC})$ with $Int(\text{Fifo}_{n-n})$ by giving an execution σ where s_σ and r_σ are both chains, but σ itself is not one. Similarly, Figure 10 separates $Int(\text{Fifo}_{n-n})$ with $Int(\text{Fifo}_{1-n})$ giving an execution σ where $r_\sigma(p_1)$ and $r_\sigma(p_2)$ are chains but not σ.
2. Here we only need to prove the separation of $Int(\text{Fifo}_{n-n})$ with $Int(\text{Fifo}_{n-1})$: Figure 10 represents an adequate execution σ where $s_\sigma(p_1)$ and $s_\sigma(p_2)$ are chains but not σ.
3. Incomparability is shown by the executions from Figure 11 (where $r_\sigma(p_1)$ is not a chain but $s_\sigma(p_1), s_\sigma(p_2)$ and $s_\sigma(p_3)$ are) and Figure 12 (where $s_\sigma(p_3)$ is not a chain, but $r_\sigma(p_1), r_\sigma(p_2)$ and $r_\sigma(p_3)$ are). ◀

4.5 Interpretation of interiors

Whereas closures expand the initial set of executions to account for additional uncertainty, interiors trim it down to introduce certainty. In interiors, the message ordering is enforced by the causal order itself, ensuring it despite the non-determinism of asynchronicity. Such certainty can be leveraged as a mean of implementation, by ensuring that a given system only generates computations in the interior of a message ordering.

Let us take $Int(\text{Fifo}_{n-n})$ as an example: due to its characterization (Theorem 24), its computations must ensure both causal ordering and that all messages simultaneously in transit are between the same pair of peers. This description fits the definition of a token network, where the peer holding the token is the only one allowed to send (to a single destination peer). It then sends the token along in its last message, and on again. Similar

characterizations can be worked out for the other interiors: Int(RSC) defines a token model where the token is sent with each message, and $Int(\text{Fifo}_{n-1})$ and $Int(\text{Fifo}_{1-n})$ respectively defines token models with a send token for each peer in one case and a receive token for each peer in the other. Thus a distributed system using any of these token models implements the corresponding interior.

One apparent contradiction with the literature is that our interiors cannot be implemented in the sense of Murty and Garg [22]. Indeed, the latter paper shows that any message ordering (over computations) implementable by an inhibitory protocol must contain a specific subset of computations, which is equivalent to our $Clos$(RSC): our interiors don't satisfy this condition. By inhibitory protocol, [22] means a protocol only able to delay both send events requested by the user and deliveries of received messages. Even if our token-based conditions do not help building an inhibitory protocol, they do implement the interiors in a different way: such systems only generate computations in the corresponding interiors, and thus executions in the corresponding orderings over executions.

5 Related Work

Message orderings over computations. The Fifo_{1-1} ordering (traditionally called FIFO) dates back to the first distributed algorithms, such as Chandy-Lamport Snapshot [5]. The latter paper exemplifies the treatment of this ordering: it is brushed aside in one sentence, not even given a proper name.

Causal ordering has a deeper history. Following the connection drawn by Lamport between Happened-Before and potential causality [20], the former was used as a basis for causal broadcast ordering by Birman et al. [4]. But the broadcast part blurred the exact definition and eased the implementation: a point-to-point formalization by Schiper and al. [24] followed. It made use notably of multiple vector clocks (independently invented by Fidge [14] and Mattern [21]) to track potential causality beyond the logical clocks introduced by Lamport [20]. Following this line of research, Schwarz and Mattern [25] motivated the general problem of detecting the needed causality, while Kshemkalyani and Singhal [19] proved necessary and sufficient conditions for implementing Causal ordering, as well as an optimal implementation based on these.

$Clos$(RSC) (usually simply called RSC) is a late invention compared to the two previous orderings: it was introduced independently by Soneoka and Ibaraki [26] and Charron-Bost et al. [8] after almost all the papers mentioned above. By being non-blocking yet allowing each message to be alone in transit, $Clos$(RSC) represents the best approximation of synchronous (or rendez-vous) communication in an asynchronous setting.

Causal ordering and $Clos$(RSC) were used for characterizing implementability of message orderings by inhibitory protocols: Murty and Garg [22] showed that an ordering must contain Causal to be implementable without control messages, and the equivalent of $Clos$(RSC) to be implementable at all. Charron-Bost et al. [8] also used these three orderings as a basis for their hierarchy.

Beyond Happened-Before. Following Birman et al. [4] definition and implementation of causal broadcast ordering, Cheriton and Skeen [10] shed light into the limitations of Happened-Before as causality. Namely, that local events are always totally ordered by the Happened-Before relation while they might be causally independent. This in turn causes additional latency: if two send events are causally independent but ordered by Happened-Before, one reception might unnecessarily wait for the other.

As a follow-up, Tarafdar and Garg [27] developed the idea of potential causality for approximating true causality in the same way Happened-Before order approximates real time.

The difference stems from the local ordering of events: whereas the Happened-Before relation totally orders them, potential causality orders local events using an applicative and thus domain-dependent criterion. It can therefore dodge false causality. Tarafdar and Garg [27] also provided an example of potential causality applied to predicate detection, a problem where false causality causes both false positives and false negatives.

Finally, Ben-Zvi and Moses [3] extended the Happened-Before relation to the synchronous case. Their Syncausality captures the ability, when communication is bounded, to detect that no message was sent at a given time by a given peer. And their Bound Guarantees captures that if one knows a message was sent and the bound of the channel, then one knows after which point the message was necessarily received.

Knowledge About Uncertainty. Both closures and interiors provide additional knowledge about the computation: whether it might follow the ordering and whether it must, respectively.

Cooper and Marzullo [12] also treat knowledge through existential and universal quantification, but this time on path of consistent cuts. They define the *Possibly* and *Definitely* operators for predicates over observations (consistent cuts): *Possibly* ϕ means that there is a path in the lattice of observations passing through an observation satisfying ϕ; *Definitely* ϕ means that all paths pass through an observation satisfying ϕ. In a subsequent paper, Charron-Bost et al. [7] showed that these two operators were temporal analogous to the local knowledge operator *Knows*. They leveraged this analogy to prove a temporal counterpart to Chandy and Misra's Knowledge Change Theorem [6].

The latter paper by Chandy and Misra [6] was part of the effort to formalize knowledge in distributed systems. As almost all concurrent and subsequent efforts, it was based on the notion of indistinguishability: a process knows a predicate valuation if and only if this valuation is constant over all possible "worlds" the process cannot distinguish. Halpern and Moses [15] anchored this possible worlds semantics with axiomatisations of different knowledge characterizations, using Kripke structures as models. The interested reader is directed to Fagin et al. [13] for a thorough treatment of the subject.

Topology in distributed systems. Lastly, our approach is based on topology. The characterization of Liveness and Safety by Alpern and Schneider [1] can be considered the first application of topology to distributed computing: although their topology on the execution space is not specific to distributed systems, their characterizations play a significant role in proving correctness and efficiency of distributed algorithms. A version taking into account failures was subsequently proposed by Charron-Bost et al. [9].

Another application of topology concerns wait-freeness. A wait-free algorithm, following Herlihy's definition [16], is one where each process must terminate within a finite number of steps. Waiting for another process is thus not possible, which makes wait-free algorithms a powerful tool for fault-tolerance. Herlihy and Shavit [18] applied algebraic topology to the problem of solving tasks in a wait-free way. They showed that both the possible input and output of a task could be represented as simplicial complexes, mathematical structures from algebraic topology. Then the possibility or impossibility of wait-freely solving a task can be rephrased as the existence or nonexistence of a specific map from the input complex to the output one. We direct the interested reader to Herlihy et al. [17] for an ample treatment of this applications of algebraic topology to distributed computability.

6 Conclusion and Perspectives

Through our topology, we offered an analysis of asynchronous message orderings over executions. Their closures precisely characterize the additional uncertainty brought by asynchronicity, allowing us to show that orderings are indistinguishable at the level of computations. Interiors, on the other hand, characterize the necessary reduction to ensure the message ordering despite the non-determinism of asynchronicity. This in turn can be used as an operational constraint for implementing the ordering.

Far from putting an end to this line of research, we feel this work opens up interesting directions for further inquiry.

- First, finding a characterization for $Clos(\text{Fifo}_{n-n})$ will precise the distinguishability condition between message orderings, as well as give another tool for distributed systems programmers to decide the ordering they need.
- Multicast and broadcast message orderings are also predicates over computations and executions. Thus the topological approach introduced in this paper can be applied to them too.
- The interpretation of interiors brought up the question of implementation. Is there a more general definition than the existence of an inhibitory protocol? If so, then it will need to account for our implementations through system-level constraints such as tokens.
- Message orderings over executions which do not collapse ask for a stronger order than the causal one, thus more knowledge about the ordering of events. Studying this additional knowledge will precise the respective power of message orderings.

References

1 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

2 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchronously communicating systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2012. doi:10.1007/978-3-642-27940-9_5.

3 Ido Ben-Zvi and Yoram Moses. Beyond lamport's *Happened-before*: On time bounds and the ordering of events in distributed systems. *J. ACM*, 61(2):13:1–13:26, 2014. doi:10.1145/2542181.

4 Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987. doi:10.1145/7351.7478.

5 K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. doi:10.1145/214451.214456.

6 K. Mani Chandy and Jayadev Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986. doi:10.1007/BF01843569.

7 Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst.*, 17(1):157–179, 1995. doi:10.1145/200994.201005.

8 Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996. doi:10.1007/s004460050018.

9 Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency*

Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings, volume 1877 of *Lecture Notes in Computer Science*, pages 552–565. Springer, 2000. doi:10.1007/3-540-44618-4_39.

10 David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 27(5):44–57, 1993. doi:10.1145/173668.168623.

11 Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. On the diversity of asynchronous communication. *Formal Asp. Comput.*, 28(5):847–879, 2016. doi:10.1007/s00165-016-0379-x.

12 Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *SIGPLAN Not.*, 26(12):167–174, dec 1991. doi:10.1145/127695.122774.

13 Ronald Fagin, Joseph Y. Halpern, Moshe Y. Vardi, and Yoram Moses. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 1995.

14 Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

15 Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990. doi:10.1145/79147.79161.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

17 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 1st edition, 2013.

18 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

19 Ajay D. Kshemkalyani and Mukesh Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11(2):91–111, 1998. doi:10.1007/s004460050044.

20 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

21 Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

22 Venkatesh V. Murty and Vijay K. Garg. Characterization of message ordering specifications and protocols. In *Proceedings of the 17th International Conference on Distributed Computing Systems, Baltimore, MD, USA, May 27-30, 1997*, pages 492–499. IEEE Computer Society, 1997. doi:10.1109/ICDCS.1997.603392.

23 Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, 1991. doi:10.1016/0020-0190(91)90008-6.

24 André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In Jean-Claude Bermond and Michel Raynal, editors, *Distributed Algorithms, 3rd International Workshop, Nice, France, September 26-28, 1989, Proceedings*, volume 392 of *Lecture Notes in Computer Science*, pages 219–232. Springer, 1989. doi:10.1007/3-540-51687-5_45.

25 Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994. doi:10.1007/BF02277859.

26 Terunao Soneoka and Toshihide Ibaraki. Logically instantaneous message passing in asynchronous distributed systems. *IEEE Trans. Computers*, 43(5):513–527, 1994. doi:10.1109/12.280800.

27 Ashis Tarafdar and Vijay K. Garg. Addressing false causality while detecting predicates in distributed programs. In *Proceedings of the 18th International Conference on Distributed*

Computing Systems, Amsterdam, The Netherlands, May 26-29, 1998, pages 94–101. IEEE Computer Society, 1998. doi:10.1109/ICDCS.1998.679491.

28 William T. Trotter. *Combinatorics and Partially Ordered Sets*. The Johns Hopkins University Press, 1992.

A Proof of the causal knot criterion for Fifo_{1-n}

We first introduce $\prec^{\sigma}_{1-n}$, which corresponds to $\prec^{\sigma}_{c}$ augmented with the constraints of Fifo_{1-n}.

▶ **Definition 28** (1-N order). Let $\sigma \in Exec$ and Σ the underlying set of events. Then $\lhd^{\sigma}_{1-n} \triangleq \{(r(m_1), r(m_2)) \in \Sigma \times \Sigma \mid s(m_1) \prec^{\sigma}_{p} s(m_2)\}$. And $\prec^{\sigma}_{1-n} \triangleq \prec^{\sigma}_{c} \cup \lhd^{\sigma}_{1-n}$.

Then the next Lemma reduces the membership of an execution σ in $Clos(\text{Fifo}_{1-n})$ to whether or not $\prec^{\sigma}_{1-n}$ contains a cycle.

▶ **Lemma 29.** *Let $\sigma \in Exec$. Then $\sigma \in Clos(Fifo_{1-n}) \iff \prec^{\sigma}_{1-n}$ is antisymmetric.*

Proof. ($\Rightarrow$) We show the contrapositive: If $\prec^{\sigma}_{1-n}$ contains a cycle, then $\sigma \notin Clos(\text{Fifo}_{1-n})$.

Let $\sigma \in Exec$ such that $\prec^{\sigma}_{1-n}$ contains a cycle. Then its transitive closure is not a partial order: it has no linear extensions. From that fact, we now prove that no execution in $[\sigma]_{\equiv_c}$ satisfies Fifo_{1-n}, and thus that $\sigma \notin Clos(\text{Fifo}_{1-n})$.

Let $\sigma' \equiv_c \sigma$. By our hypothesis that $\prec^{\sigma}_{1-n}$ contains a cycle, we have $\prec^{\sigma}_{1-n} \not\subset \prec^{\sigma'}$. But since $\sigma' \equiv_c \sigma$, we have $\prec^{\sigma}_{c} \subset \prec^{\sigma'}$. We thus conclude that $\lhd^{\sigma}_{1-n} \not\subset \prec^{\sigma'}$. Thus $\exists m_1, m_2$ such that $s(m_1) \prec^{\sigma}_{p} s(m_2) \wedge r(m_1) \not\prec^{\sigma'} r(m_2)$. By totality of $\prec^{\sigma'}$, we have $s(m_1) \prec^{\sigma}_{p} s(m_2) \wedge r(m_2) \prec^{\sigma'} r(m_1)$, and thus σ' violates Fifo_{1-n}.

($\Leftarrow$) Let $\sigma \in Exec$ such that $\prec^{\sigma}_{1-n}$ is antisymmetric. Then the transitive closure of $\prec^{\sigma}_{1-n}$ is reflexive (because $\prec^{\sigma}_{c}$ is reflexive), transitive and antisymmetric. It is therefore a partial order, who has a linear extension agreeing both with $\prec^{\sigma}_{c}$ and $\lhd^{\sigma}_{1-n}$. This linear extension defines an execution $\sigma' \equiv_c \sigma$ (because $\prec^{\sigma}_{c} \subset \prec^{\sigma'}$) such that σ' satisfies Fifo_{1-n} (because $\lhd^{\sigma}_{1-n} \subset \prec^{\sigma'}$).

We conclude that $\sigma \in Clos(\text{Fifo}_{1-n})$. ◀

We can now prove the causal knot criterion for Fifo_{1-n}, which characterizes $Clos(\text{Fifo}_{1-n})$ by the absence of causal knots.

▶ **Theorem 30** (Causal Knot Criterion for Fifo_{1-n}). *Let $\sigma \in Exec$. Then $\sigma \in Clos(\text{Fifo}_{1-n})$ $\iff$ σ contains no causal knot.*

Proof. By Lemma 29 and the transitivity of equivalence, we only need to prove that $\prec^{\sigma}_{1-n}$ is antisymmetric $\iff$ σ contains no causal knot.

($\Rightarrow$) We prove the contrapositive: if σ contains a causal knot, then $\prec^{\sigma}_{1-n}$ is not antisymmetric.

Let $\sigma \in Exec$ with a causal knot of size k. By definition of causal knots, there are $2k$ messages $m_1, .., m_{2k}$ such that $\iff \forall i \in [1, 2k] : \begin{pmatrix} i \equiv 1 \bmod 2 : s(m_i) \prec^{\sigma}_{c} s(m_{i+1}) \\ i \equiv 0 \bmod 2 : r(m_i) \prec^{\sigma}_{c} r(m_{i+1}) \end{pmatrix}$, where $m_{2k+1} = m_1$. We now define another sequence of messages $m'_1, .., m'_{2k}$ from the messages of the causal knot: $\forall i \in [1, 2k]$:

$$\begin{pmatrix} i \equiv 1 \bmod 2 : if\ s(m_i) \prec^{\sigma}_{p} s(m_{i+1}) \\ \qquad\qquad\quad then\ m'_i = m_i \\ \qquad\qquad\quad else\ m'_i = m, where\ m \in MES : s(m_i) \prec^{\sigma}_{p} s(m) \prec^{\sigma}_{c} r(m) \prec^{\sigma}_{c} s(m_{i+1}) \\ i \equiv 0 \bmod 2 : m'_i = m_i \end{pmatrix},$$

where $m'_{2k+1} = m'_1$. The case $i \equiv 0 \bmod 2$ is well-defined since the causal order corresponds to either the peer order or a chain of messages. We then have $\forall i \in [1, 2k]$: $\begin{pmatrix} i \equiv 1 \bmod 2 : r(m'_i) \lhd^\sigma_{1-n} r(m'_{i+1}) \\ i \equiv 0 \bmod 2 : r(m'_i) \prec^\sigma_c r(m'_{i+1}) \end{pmatrix}$, where $m'_{2k+1} = m'_1$. Thus $\prec^\sigma_{1-n}$ contains a cycle and is not antisymmetric.

($\Leftarrow$) We prove the contrapositive: if $\prec^\sigma_{1-n}$ contains a cycle, then σ contains a causal knot.

Let $\sigma \in Exec$ such that $\prec^\sigma_{1-n}$ contains a cycle. Since both $\prec^\sigma_c$ and $\lhd^\sigma_{1-n}$ are partial orders, they are both antisymmetric: any minimal cycle consists of an alternation of those two. Considering such a minimal cycle of size s, we can deduce by transitivity of $\prec^\sigma_c$ and $\lhd^\sigma_{1-n}$ that it is composed of s events $e_1, .., e_s$ such that $\forall i \in [1, s]$: $\begin{pmatrix} i \equiv 1 \bmod 2 : e_i \lhd^\sigma_{1-n} e_{i+1} \\ i \equiv 0 \bmod 2 : e_i \prec^\sigma_c e_{i+1} \end{pmatrix}$, where $e_{s+1} = e_1$. By minimality of the cycle and $e_1 \lhd^\sigma_{1-n} e_2$, we have $e_s \prec^\sigma_c e_1$ and thus $s \equiv 0 \bmod 2$. Let $k = s/2$.

Recall that $\lhd^\sigma_{1-n}$ only orders receptions: $e_1, .., e_s$ are thus receptions. Their messages are distinct, by minimality of the cycle. Then the previous characterization of the cycle can be stated as $\exists m_1, .., m_{2k} \in MES$ such that $\forall i \in [1, 2k]$: $\begin{pmatrix} i \equiv 1 \bmod 2 : r(m_i) \lhd^\sigma_{1-n} r(m_{i+1}) \\ i \equiv 0 \bmod 2 : r(m_i) \prec^\sigma_c r(m_{i+1}) \end{pmatrix}$, where $m'_{2k+1} = m'_1$. By definition of $\lhd^\sigma_{1-n}$, this is a causal knot of size k. ◀

B Proof of the characterizations of $Int(\text{Fifo}_{n-n})$, $Int(\text{Fifo}_{1-n})$ and $Int(\text{Fifo}_{n-1})$

B.1 Proof of preliminary lemma

The reduction from Corollary 22 allows us to prove that when two send events (respectively two receptions) are not causally ordered in an execution, there is a causally equivalent execution where these two events are inverted, whereas the corresponding communication events (the receptions for the send events and conversely) keep the same order. These inversions correspond to potential breaking points by causal equivalence for message orderings, a formalization of the intuition behind the counter-examples used in the proof of Theorem 11. They will therefore constitute our main tool for showing non-membership in a given interior.

▶ **Lemma 31** (Inversion of communication events). *Let $\sigma \in Exec$, and $m_1, m_2 \in MES$ such that $r(m_1), r(m_2) \in \sigma$. Then:*

1. $s(m_1) \prec^\sigma s(m_2) \wedge s(m_1) \not\prec^\sigma_c s(m_2)$
$\implies \exists \sigma' \equiv_c \sigma : s(m_2) \prec^{\sigma'} s(m_1) \wedge \prec^{\sigma'}_{|\{r(m_1), r(m_2)\}} = \prec^\sigma_{|\{r(m_1), r(m_2)\}}$
2. $r(m_1) \prec^\sigma r(m_2) \wedge r(m_1) \not\prec^\sigma_c r(m_2)$
$\implies \exists \sigma' \equiv_c \sigma : r(m_2) \prec^{\sigma'} r(m_1) \wedge \prec^{\sigma'}_{|\{s(m_1), s(m_2)\}} = \prec^\sigma_{|\{s(m_1), s(m_2)\}}$

Proof. Let $\sigma \in Exec$, and let $m_1, m_2 \in MES$ such that $r(m_1), r(m_2) \in \sigma$. Both cases boil down to exhibiting a $\sigma' \equiv_c \sigma$ satisfying the condition. If we had a linear extension of $\prec^\sigma_{c|\{s(m_1), s(m_2), r(m_1), r(m_2)\}}$ satisfying the condition, Lemma 22 would give us such valid σ'. We therefore prove the existence of a suitable linear extension of $\prec^\sigma_{c|\{s(m_1), s(m_2), r(m_1), r(m_2)\}}$ in each case.

1. Let $s(m_1) \prec^\sigma s(m_2)$ and $s(m_1) \not\prec^\sigma_c s(m_2)$. Let $\prec_l$ a total order on $\{s(m_1), s(m_2), r(m_1), r(m_2)\}$ defined by $s(m_2) \prec_l s(m_1) \prec_l r(m_1)$, $s(m_2) \prec_l s(m_1) \prec_l r(m_2)$ and $\prec_{l|\{r(m_1), r(m_2)\}} = \prec^\sigma_{|\{r(m_1), r(m_2)\}}$.
Since $s(m_1) \not\prec^\sigma_c s(m_2)$ by hypothesis and $\prec^\sigma_c$ is transitive, we get $r(m_1) \not\prec^\sigma_c s(m_2)$. This in turn implies $r(m_1) \not\prec^\sigma_c s(m_1) \wedge r(m_1) \not\prec^\sigma_c s(m_2)$.

On the other hand, transitivity of $\prec^\sigma$ yields $s(m_1) \prec^\sigma r(m_2)$. Thus $r(m_2) \not\prec^\sigma_c s(m_1)$, and we deduce $r(m_2) \not\prec^\sigma_c s(m_1) \wedge r(m_2) \not\prec^\sigma_c s(m_2)$.
$\prec_l$ is therefore a linear extension of $\prec^\sigma_{c|\{s(m_1),s(m_2),r(m_1),r(m_2)\}}$.

2. The proof is similar to the previous case, with $\prec_{l'}$ a total order on $\{s(m_1), s(m_2), r(m_1), r(m_2)\}$ such that $s(m_1) \prec_{l'} r(m_2) \prec_{l'} r(m_1)$, $s(m_1) \prec_{l'} r(m_2) \prec_{l'} r(m_1)$ and $\prec_{l'|\{s(m_1),s(m_2)\}} = \prec^\sigma_{|\{s(m_1),s(m_2)\}}$. ◀

B.2 Proof of the characterization of $Int(\text{Fifo}_{n-n})$

We now prove the characterization of $Int(\text{Fifo}_{n-n})$ from Theorem 24

Proof. ($\Rightarrow$) **Assume** the opposite: let $\sigma \in Int(\text{Fifo}_{n-n})$ and r_σ or s_σ is not a chain ($\sigma \in Exec(\text{Causal})$ necessarily).

- If r_σ is not a chain, $\exists m_1, m_2 \in MES$ such that $r(m_1) \prec^\sigma r(m_2)$ and $r(m_1) \not\prec^\sigma_c r(m_2)$. Since $\sigma \in Int(\text{Fifo}_{n-n})$, we get $s(m_1) \prec^\sigma s(m_2)$. The hypothesis $r(m_1) \not\prec^\sigma_c r(m_2)$ and Lemma 31 then entail the existence of $\sigma' \equiv_c \sigma$ with $s(m_1) \prec^{\sigma'} s(m_2)$ and $r(m_2) \prec^{\sigma'} r(m_1)$. But $\sigma' \notin Exec(\text{Fifo}_{n-n})$. Hence $\sigma \notin Int(\text{Fifo}_{n-n})$. **Contradiction**.
- If s_σ is not a chain, the same application of Lemma 31 to the case $s(m_1) \prec^\sigma s(m_2)$, $s(m_1) \not\prec^\sigma_c s(m_2)$ and $r(m_1) \prec^\sigma r(m_2)$ gives us a $\sigma'' \equiv_c \sigma$ with $\sigma'' \notin Exec(\text{Fifo}_{n-n})$. Hence $\sigma \notin Int(\text{Fifo}_{n-n})$. **Contradiction**.

($\Leftarrow$) We prove the contrapositive, namely: $\sigma \notin Int(\text{Fifo}_{n-n}) \implies \sigma \notin Exec(\text{Causal}) \vee s_\sigma$ not a chain $\vee r_\sigma$ not a chain.

Let $\sigma \in Exec$ such that $\sigma \notin Int(\text{Fifo}_{n-n})$. Thus $\exists \sigma' \equiv_c \sigma$ such that $\exists m_1, m_2 \in MES$: $s(m_1) \prec^{\sigma'} s(m_2) \wedge r(m_2) \prec^{\sigma'} r(m_1)$. We then have 2 cases.

- $s(m_1) \prec^\sigma_c s(m_2) \wedge r(m_2) \prec^\sigma_c r(m_1)$. Then by definition of the causal order, either $r(m_2) \prec^\sigma_p r(m_1)$ or $\exists m \in MES$ such that $s(m_1) \prec^\sigma_c s(m_2) \prec^\sigma_c r(m_2) \prec^\sigma_c s(m) \prec^\sigma_c r(m) \prec^\sigma_p r(m_1)$. Either way, we conclude $\sigma \notin Exec(\text{Causal})$.
- $s(m_1) \not\prec^\sigma_c s(m_2) \vee r(m_2) \not\prec^\sigma_c r(m_1)$. Then s_σ or r_σ is not a chain. ◀

B.3 Proof of the characterization of $Int(\text{Fifo}_{n-1})$ and $Int(\text{Fifo}_{1-n})$

We now prove both characterizations of $Int(\text{Fifo}_{n-1})$ and $Int(\text{Fifo}_{1-n})$ from Theorem 25 and Theorem 26 respectively.

Proof. ($\Rightarrow$) **Assume** the opposite: let $\sigma \in Int(\text{Fifo}_{n-1})$ and $p \in PEERS$ such that $s_\sigma(p)$ is not a chain ($\sigma \in Exec(\text{Causal})$ necessarily).

Thus $\exists m_1, m_2 \in MES$ such that $s(m_1), s(m_2) \in s_\sigma(p)$, $s(m_1) \prec^\sigma s(m_2)$ and $s(m_1) \not\prec^\sigma_c s(m_2)$. Since $\sigma \in Exec(\text{Fifo}_{n-1})$ and $peer(r(m_1)) = peer(r(m_2))$, we have $r(m_1) \prec^\sigma_p r(m_2)$. Then Lemma 31 implies $\exists \sigma' \equiv_c \sigma$ such that $s(m_2) \prec^{\sigma'} s(m_1), r(m_1) \prec^{\sigma'} r(m_2)$ and $peer(r(m_1)) = peer(r(m_2))$. But $\sigma' \notin Exec(\text{Fifo}_{n-1})$, and thus $\sigma \notin Int(\text{Fifo}_{n-1})$. **Contradiction**.

($\Leftarrow$) We prove the contrapositive, namely: $\sigma \notin Int(\text{Fifo}_{n-1}) \implies \sigma \notin Exec(\text{Causal}) \vee \exists p \in PEERS : s_\sigma(p)$ not a chain.

Let $\sigma \in Exec$ such that $\sigma \notin Int(\text{Fifo}_{n-1})$. Thus $\exists \sigma' \equiv_c \sigma$ such that $\exists m_1, m_2 \in MES$: $s(m_1) \prec^{\sigma'} s(m_2) \wedge r(m_2) \prec^\sigma_p r(m_1)$. We then have 2 cases.

- $s(m_1) \prec^\sigma_c s(m_2)$. From $s(m_1) \prec^\sigma_c s(m_2) \prec^\sigma_c r(m_2) \prec^\sigma_p r(m_1)$, we conclude $\sigma \notin Exec(\text{Causal})$.
- $s(m_1) \not\prec^\sigma_c s(m_2)$. Then $s_\sigma(peer(r(m_1)))$ is not a chain. ◀

Proof. ($\Rightarrow$) **Assume** the opposite: let $\sigma \in Int(\text{Fifo}_{1-n})$ and $p \in PEERS$ such that $r_\sigma(p)$ is not a chain ($\sigma \in Exec(\text{Causal})$ necessarily).

Thus $\exists m_1, m_2 \in MES$ such that $r(m_1), r(m_2) \in r_\sigma(p)$, $r(m_1) \prec^\sigma r(m_2)$ and $r(m_1) \not\prec_c^\sigma r(m_2)$. Since $\sigma \in Exec(\text{Fifo}_{1-n})$ and $peer(s(m_1)) = peer(s(m_2))$, we have $s(m_1) \prec_p^\sigma s(m_2)$. Then Lemma 31 implies $\exists \sigma' \equiv_c \sigma$ such that $r(m_2) \prec^{\sigma'} r(m_1), s(m_1) \prec^{\sigma'} s(m_2)$ and $peer(s(m_1)) = peer(s(m_2))$. But $\sigma' \notin Exec(\text{Fifo}_{1-n})$, and thus $\sigma \notin Int(\text{Fifo}_{1-n})$. **Contradiction**.

($\Leftarrow$) We prove the contrapositive, namely: $\sigma \notin Int(\text{Fifo}_{1-n}) \implies \sigma \notin Exec(\text{Causal}) \vee \exists p \in PEERS : r_\sigma(p)$ not a chain.

Let $\sigma \in Exec$ such that $\sigma \notin Int(\text{Fifo}_{1-n})$. Thus $\exists \sigma' \equiv_c \sigma$ such that $\exists m_1, m_2 \in MES : s(m_1) \prec_p^\sigma s(m_2) \wedge r(m_2) \prec^{\sigma'} r(m_1)$. We then have 2 cases.

- $r(m_2) \prec_c^\sigma r(m_1)$. Then either $r(m_2) \prec_p^\sigma r(m_1)$ or $\exists m \in MES : r(m_2) \prec_c^\sigma s(m) \prec_c^\sigma r(m) \prec_p^\sigma r(m_1)$, by definition of the causal order. Both cases yield $\sigma \notin Exec(\text{Causal})$.
- $r(m_2) \not\prec_c^\sigma r(m_1)$. Since $r(m_2) \prec^{\sigma'} r(m_1)$ gives us $r(m_1) \not\prec_c^\sigma r(m_2)$, we conclude that $r_\sigma(peer(s(m_1)))$ is not a chain. ◀

Constant Space and Non-Constant Time in Distributed Computing

Tuomo Lempiäinen[1] **and Jukka Suomela**[2]

1 **Department of Computer Science, Aalto University, Espoo, Finland**
`tuomo.lempiainen@aalto.fi`
2 **Department of Computer Science, Aalto University, Espoo, Finland**
`jukka.suomela@aalto.fi`

Abstract

While the relationship of time and space is an established topic in traditional centralised complexity theory, this is not the case in distributed computing. We aim to remedy this by studying the time and space complexity of algorithms in a weak message-passing model of distributed computing. While a constant number of communication rounds implies a constant number of states visited during the execution, the other direction is not clear at all. We show that indeed, there exist non-trivial graph problems that are solvable by constant-space algorithms but that require a non-constant running time. Somewhat surprisingly, this holds even when restricted to the class of only cycle and path graphs. Our work provides us with a new complexity class for distributed computing and raises interesting questions about the existence of further combinations of time and space complexity.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.3 Complexity Measures and Classes

Keywords and phrases distributed computing, space complexity, constant-space algorithms, weak models, Thue–Morse sequence

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.30

1 Introduction

In the classical centralised theory of computing, the study of space-limited computation has helped us with understanding computability and computational complexity in general:

1. Constant-space models (finite-state machines) provide a very well-understood solid foundation.
2. Space-limited complexity classes (e.g., PSPACE) can be successfully related with time-limited complexity classes (e.g., NP $\subseteq$ PSPACE $\subseteq$ EXP).

In this work, we use similar ideas in the study of distributed computing, in particular, in the context of distributed graph algorithms.

Networks of finite-state machines. The natural distributed analogue of a deterministic finite-state machine is a *network* of deterministic finite-state machines. For brevity, we call distributed algorithms that use only finitely many states per node *constant-space algorithms*. See Section 2 for a proper definition of the model of computation.

Variants of this setting have been studied in many papers, but perhaps the most elementary question related to distributed graph algorithms has not been answered yet:

> Can we say anything about the *time* complexity of graph problems that are solvable with a network of finite-state machines?

Naturally, many constant-time algorithms are also constant-space algorithms, but does the converse hold in general?

Key challenges. At first, the question may seem trivial: a path of n finite-state machines can simulate the computation of any Turing machine with n units of space. We can easily construct algorithms that take time that is exponential in n to terminate; a simple example is a binary counter that counts from all-0 to all-1.

However, this does not imply that there are graph problems that are solvable in constant space but not in constant time! There are two main obstacles:

1. The fact that there exists a slow algorithm does not imply that the problem cannot be solved with a fast algorithm. In the binary counter example, we produce an all-1 output if we have an all-0 input; this is of course a trivial problem to solve in constant time.
2. We exploit a *promise*: we assume that the input is a path.

The second obstacle may at first seem like a mere technicality, but it turns out to be the key difference that separates the centralised computational complexity theory from its distributed counterpart when we consider constant-space machines.

In centralised computing, the input is by definition a string with a well-defined starting point and endpoint. A natural analogue of this in distributed computing is a (directed) path. If we have a promise that the input is a path, we can easily construct examples of problems that are solvable with constant-space but not with constant-time algorithms, by following ideas that work in the case of centralised computing:

- Classical centralised language: "Recognise the language of strings that have an even number of characters."
- Distributed graph problem: "If the input is a path with an even number of nodes, all nodes have to output 1, otherwise all nodes have to output 0."

However, in distributed graph algorithms, the input is a graph that comes from an adversary (see Section 2.4), and the graph may be highly symmetric. A prime example is a cycle graph; indeed, breaking symmetry in cycles is one of the most classical problems in the area of distributed graph algorithms.

Informally, centralised finite-state machines never get stuck in infinite loops, as eventually the input string that they are scanning will end, but a distributed algorithm might easily fail to terminate if the input is a cycle. The usual escape is to assume that we have globally unique node identifiers (or access to randomness), but non-constant-size identifiers are not a natural assumption with constant-space algorithms.

Indeed, if we insist that the algorithm is a well-defined algorithm in the sense that it always terminates, it is difficult to see if we can solve *anything* non-trivial with constant-space algorithms without resorting to some assumption on the input: all nodes have to terminate in some finite time T even if the input is a cycle, and a cycle is indistinguishable from a sufficiently long path, and hence all nodes have to terminate in fixed time T in arbitrarily long paths. It would be tempting to conjecture that constant space (and well-defined algorithms without any promise) would necessarily imply constant time. In this work, we show that this is not the case.

Contributions. We give a counterexample that separates constant-space and constant-time algorithms (see Section 4). More precisely, we construct a well-defined graph problem Π such that:

- Π is solvable with constant-space algorithms (even if we consider a very restricted subset of constant-space machines),
- Π cannot be solved in $o(n)$ time with any distributed algorithm (even if we use a much stronger model of computation, e.g., randomised algorithms in the LOCAL model).

Here Π is a well-defined input–output problem in the classical sense: for every labelled input graph G we have a set $\Pi(G)$ of feasible output labellings. We only need constant-size input and output labels, and our result holds even if we restrict to input graphs of maximum degree at most 2.

The key technical challenge here is to engineer a problem Π and a constant-space algorithm $\mathcal{A}$ for Π such that for *any* input graph G, with *any* input labelling, algorithm $\mathcal{A}$ is guaranteed to stop in finite time; yet Π has to be sufficiently non-trivial so that it cannot be solved in constant time – not even in graphs of maximum degree 2. Here we resort to so-called Thue–Morse sequence as a source of inspiration (see Section 2.5).

Open questions. We emphasise that our problem Π is a purely artificial problem that only serves the purpose of separating constant-space and constant-time algorithms. Furthermore, many problems that have been extensively studied in the area of distributed graph algorithms are LCL (locally checkable labelling) problems [20], and our problem Π is not a member of this problem family. Hence there are immediate follow-up questions that we leave for future work:

1. Does there exist a *natural* graph problem that separates constant space and constant time?
2. Does there exist an LCL problem that separates constant space and constant time?

Related work. While the relationship between space and time complexity in distributed computing is a rather novel topic, various time complexity classes and the relationships between them have been studied a lot in many of the established works of the field [14, 16, 19, 20] and also very recently [3, 4, 5, 6, 7, 13].

The line of research on anonymous models of distributed computing was initiated by Angluin [2], who introduced the well-known port numbering model. Our model of computation can be seen as a further restriction of the port-numbering model, with port numbers stripped out. While port numbers are a natural assumption in wired networks, the weaker variant makes more sense when applying distributed computing to a wireless setting. While our model has not been studied that much in prior work, a so-called *beeping model* [1, 9] is essentially similar. In the hierarchy of seven models defined by Hella et al. [15], our model is the weakest one; they call it the SB model. On the other hand, the case where nodes receive messages in a multiset instead of a set is discussed more often in the literature [15].

From the constant-space point of view, our setting bears similarities to the field of cellular automata [12, 23, 24]. In a cellular automaton, each cell can be in one of a constant number of states, and each cell updates its state synchronously using the same rule. However, in the case of cellular automata, one is usually interested in the kind of patterns an automata converges in, while we require each node to eventually stop and produce an output.

On the side of distributed computing, Emek and Wattenhofer [11] have considered a model where the network consists of finite-state machines – hence making the space complexity constant. However, their model is asynchronous and randomised, while we study a fully

synchronous and deterministic setting. Recently, also Kuusisto and Reiter [18, 21, 22] have considered networks of finite-state machines, but their machines either halt in constant time [21] or continue running indefinitely [18, 22]. Furthermore, constant memory has been studied in the setting where a set of mobile agents explores a graph [8, 10].

One of the main ingredients of our work, the Thue–Morse sequence, was used previously by Kuusisto [17], who proved that there exists a distributed algorithm that always halts in the class of graphs of maximum degree two but features a non-constant running time. However, his algorithm has also a non-constant space complexity – this is where our work provides a significant improvement.

2 Preliminaries

In this section, we define our model of computation and introduce notions needed later in the proofs.

2.1 Model of computation

We consider a model of computation where each node of a graph is a computational unit. The same graph is a communication network and an input to the algorithm. Adjacent nodes communicate with each other in synchronous rounds, and eventually each node outputs its own part of the output. All the nodes run the same deterministic algorithm. In our model, the nodes are anonymous, that is, they do not have access to unique identifiers, and furthermore, they cannot distinguish between their neighbours – they broadcast the same message to everyone and they receive the incoming messages in a set. Our model can be seen as a very weak variant of the standard LOCAL model of distributed computing.

Running time is defined as the number of communication rounds until all the nodes have halted, while space usage is defined as the number of bits per node needed to represent all the states of the algorithm. The complexity measures are considered as a function of n, the number of nodes in the graph. The amount of local computation in each round is not limited, nor is the size of messages (but constant space complexity implies that they are also constant).

In the following, we define distributed algorithms as state machines. Given an input graph, each node of the graph is equipped with an identical state machine. The graph is always assumed to be simple, finite, connected and undirected, unless stated otherwise.

At the beginning, each state machine is only aware of its own local input (taken from some fixed finite set) and the degree of the node on which it sits. Then, computation is executed in synchronous rounds. In each round, each machine

1. broadcasts a message to its neighbours,
2. receives a set of messages from its neighbours,
3. moves to a new state based on the received messages and its previous state.

Each machine is required to eventually reach one of special halting states and stop execution. The local output is then the state of the node at the time of halting.

Note that while our model of computation is rather weak, it only makes our results stronger by limiting the capabilities of algorithms. Unique identifiers or unrestricted local inputs would not make much sense in the constant-space setting. Crucially, we require nodes to always halt in any input graph. Engineering constant-space non-constant-time problems

would be quite straightforward – even in the case of degree 2 – if nodes were allowed to run indefinitely or had the ability to continue passing messages after announcing an output.[1]

Similarly, randomisation would weaken the results: with an unlimited supply of random bits, one can acquire unique identifiers with a high probability, and even a single random bit per node would allow us to construct a simple constant-space algorithm that halts with a high probability in all paths and cycles, but exhibits a non-constant running time in the worst case.

Next, we give a more formal definition of the model of computation used in this work by defining algorithms and graph problems.

2.2 Notation and terminology

For $k \in \mathbb{N}$, we denote by $[k]$ the set $\{1, 2, \ldots, k\}$. Given a graph $G = (V, E)$, the set of neighbours of a node $v \in V$ is denoted by $N(v) = \{u \in V : \{v, u\} \in E\}$. For $r \in \mathbb{N}$, the *radius-r neighbourhood* of a node $v \in V$ is $\{u \in V : \operatorname{dist}(u, v) \leq r\}$, that is, the set of nodes u such that there is a path of length at most r between v and u. We call any induced subgraph of G that contains node v simply a *neighbourhood* of v.

We will also work with strings of letters. Given a finite alphabet Σ, we denote elements of the alphabet by lowercase symbols such as $x \in \Sigma$ or $y \in \Sigma$. On the other hand, *words* (finite sequences of letters) are denoted by uppercase symbols, e.g., $X = x_1 x_2 \ldots x_i \in \Sigma^*$. Given words $X = x_1 x_2 \ldots x_i$ and $Y = y_1 y_2 \ldots y_j$, we write their concatenation simply $XY = x_1 x_2 \ldots x_i y_1 y_2 \ldots y_j$. A word X is a *subword* of Y if $Y = Y_1 X Y_2$ for some (possibly empty) words Y_1 and Y_2. We identify words of length 1 with the letter they consist of. For any letter x and $i \in \mathbb{N}$, x^i denotes the word consisting of i consecutive letters x. We say that a word X is *of the form* $x+$ if $X = x^i$ for some $i \in \mathbb{N}_+$.

2.3 Algorithms as state machines

Let $G = (V, E)$ be a graph. An *input* for G is a function $f \colon V \to I$, where I is a finite set. For each node $v \in V$, we call $f(v) \in I$ the *local input* of v.

A *distributed state machine* is a tuple $\mathcal{A} = (S, H, \sigma_0, M, \mu, \sigma)$, where

- S is a set of states,
- $H \subseteq S$ is a finite set of halting states,
- $\sigma_0 \colon \mathbb{N} \times I \to S$ is an initialisation function,
- M is a set of possible messages,
- $\mu \colon S \to M$ is a function that constructs the outgoing messages,
- $\sigma \colon S \times \mathcal{P}(M) \to S$ is a function that defines the state transitions, so that $\sigma(h, \mathcal{M}) = h$ for each $h \in H$ and $\mathcal{M} \in \mathcal{P}(M)$.

Given a graph G, and input f for G and a distributed state machine $\mathcal{A}$, the *execution* of $\mathcal{A}$ on (G, f) is defined as follows. The state of the system in round $r \in \mathbb{N}$ is a function $x_r \colon V \to S$, where $x_r(v)$ is the state of node v in round r. To begin the execution, set $x_0(v) = \sigma_o(\deg(v), f(v))$ for each node $v \in V$. Then, let $A_{r+1}(v) = \{\mu(x_r(u)) : u \in N(v)\}$

[1] Consider the following problem: nodes with local input 0 always output 0, while nodes with local input 1 output the shortest distance modulo 2 to either a degree-1 node or to a node (possibly the node itself in the case of a cycle) with local input 1. Now, nodes with local input 0 can announce their output immediately (regardless of the existence of nodes with input 1), while nodes with local input 1 have to potentially wait for a linear number of rounds.

denote the set of messages received by node v in round $r+1$. Now the new state of each node $v \in V$ is defined by setting $x_{r+1}(v) = \sigma(x_r(v), A_{r+1}(v))$.

The *running time* of $\mathcal{A}$ on (G, f) is the smallest $t \in \mathbb{N}$ for which $x_t(v) \in H$ holds for all $v \in V$. The output of $\mathcal{A}$ on (G, f) is then $x_t \colon V \to H$, where t is the running time, and for each $v \in V$, the *local output* of v is $x_t(v)$. The *space usage* of $\mathcal{A}$ on (G, f) is defined as

$$\lceil \log |\{s \in S : s = x_r(v) \text{ for some } r \in \mathbb{N}, v \in V\}| \rceil,$$

that is, the number of bits needed to encode all the states that the state machine visits in at least one node of G during the execution. In case the execution does not halt, the running time can be defined to be ∞, and if the number of visited states grows arbitrarily large, we take the space usage to also be ∞.

From now on, we will use the terms *algorithm* $\mathcal{A}$ and *distributed state machine* $\mathcal{A}$ interchangeably, implying that each algorithm can be defined formally as a state machine.

2.4 Graph problems

The computational problems that we consider are graph problems with local input – that is, the problem instance is identical to the communication network, but possibly with an additional input value given to each node.

More formally, let I and O be finite sets. A *graph problem* is a mapping $\Pi_{I,O}$ that maps each graph $G = (V, E)$ and input $f \colon V \to I$ to a set $\Pi_{I,O}(G, f)$ of valid solutions. Each solution S is a function $S \colon V \to O$. In case of *decision graph problems*, we set $O = \{\mathsf{yes}, \mathsf{no}\}$.

If $\Pi_{I,O}$ is a graph problem, $T, U \colon \mathbb{N} \to \mathbb{N}$ are functions, $\mathcal{A}$ is a distributed state machine and $\mathcal{G}$ is a class of graphs, we say that $\mathcal{A}$ *solves* $\Pi_{I,O}$ *in class* $\mathcal{G}$ *in time* T *and in space* U if for each graph $G = (V, E) \in \mathcal{G}$ and each input $f \colon V \to I$ we have that the running time of $\mathcal{A}$ on (G, f) is at most $T(|V|)$, the space usage of $\mathcal{A}$ on (G, f) is at most $U(|V|)$ and the output of $\mathcal{A}$ on (G, f) is in the set $\Pi_{I,O}(G, f)$. In that case, we also say that the *time complexity* of algorithm $\mathcal{A}$ in $\mathcal{G}$ is T and the *space complexity* of $\mathcal{A}$ in $\mathcal{G}$ is U. If $\mathcal{G}$ is omitted, it is assumed to be the class of all (simple, finite, connected and undirected) graphs.

We define the *time complexity* of a problem $\Pi_{I,O}$ in a graph class $\mathcal{G}$ to be the slowest-growing function $T \colon \mathbb{N} \to \mathbb{N}$ such that there exists an algorithm $\mathcal{A}$ that solves $\Pi_{I,O}$ in $\mathcal{G}$ in time T. The *space complexity* of a problem is defined analogously. Again, we omit $\mathcal{G}$ when it is the class of all graphs. Notice that when considering a lower bound for the complexity of a problem, the smaller the class $\mathcal{G}$, the stronger the result, while for upper bounds, the situation is reverse.

Some typical examples of graph problems considered in the distributed setting are the minimum vertex cover problem and the maximal independent set problem, where for each valid solution S, $S(v) = 1$ indicates that node v is part of the vertex cover or independent set, respectively, while $S(v) = 0$ indicates that it is not part of it. In the setting used in this work, where nodes are allowed to receive local inputs, some natural examples are the problem of outputting the largest input value among each node's neighbours, or the problem of verifying that a colouring given as an input is proper.

2.5 Thue–Morse sequence

In this section, we present a concept that will be central in the proof of our main result in Section 4. The Thue–Morse sequence is the infinite binary sequence defined recursively as follows:

▶ **Definition 1.** The *Thue–Morse* sequence is the sequence (t_i) satisfying $t_0 = 0$, and for each $i \in \mathbb{N}$, $t_{2i} = t_i$ and $t_{2i+1} = 1 - t_i$.

Thus, the beginning of the Thue–Morse sequence is

$$0110100110010110100 1 \ldots$$

For our purposes, the following two recursive definitions will be very useful.

▶ **Definition 2.** Let $T_0 = 0$. For each $i \in \mathbb{N}_+$, let $T_i = T_{i-1}\mathcal{C}(T_{i-1})$, where $\mathcal{C}$ denotes the Boolean complement.

Note that for each $i \in \mathbb{N}$, the word T_i is the prefix of length 2^i of the Thue–Morse sequence.

▶ **Definition 3.** Let $T'_0 = 0$. For each $i \in \mathbb{N}_+$, let T'_i be obtained from T'_{i-1} by substituting each occurrence of 0 with 01 and each occurrence of 1 with 10.

Again, we have $T'_0 = 0$, $T'_1 = 01$, $T'_2 = 0110$, $T'_3 = 01101001$ and so on. A straightforward induction shows that the above two definitions are equivalent: $T_i = T'_i$ for each $i \in \mathbb{N}$. We call T_i the *Thue–Morse word of length* 2^i.

The Thue–Morse sequence contains lots of squares, that is, subwords of the form XX, where $X \in \{0,1\}^*$. Interestingly, it does not contain any cubes – subwords of the form XXX. Note also that for each $i \in \mathbb{N}_+$, T_{2i} is a palindrome.

3 Warm-up: graphs of maximum degree 3

In this section, we present a graph problem that exhibits the combination of constant space complexity and non-constant time complexity, in case we do not restrict ourselves to paths and cycles. The proof is quite straightforward, which emphasises the fact that the degree-2 case considered in Section 4 is the most interesting one. We emphasise that in the following theorem, we do not need to make any additional assumptions about the graph; the described algorithm halts in all finite input graphs.

▶ **Theorem 4.** *There exist a graph problem* Π *with constant space complexity and* $\Theta(\log n)$ *time complexity. Furthermore, the time complexity of* Π *is* $\Theta(\log n)$ *also in the class of graphs of maximum degree at most 3.*

Proof. Consider the following transformation: given a *directed* graph $G' = (V', E')$ with edges labelled "L" or "R", replace each edge $e = (u, v)$ by a gadget as represented in Figure 1. The end result is an undirected graph $G = (V, E)$, where the gadgets encode the edge directions and labels of the original graph. We say that an undirected graph G is *good* if it is obtained by the above transformation from some directed binary pseudotree G' (that is, a connected directed graph where each node has outdegree at most one and indegree either 0 or 2) such that each node with incoming edges has one incoming edge labelled with "L" and one with "R".

We call a leaf node of a good graph *original*, if it does not belong to any gadget. Now, consider the following graph problem: if the input graph is good, each node has to output the parity of its distance to the nearest original leaf node; if not, each node is allowed to output anything. In any case, all the nodes have to halt.

Define an algorithm $\mathcal{A}$ as follows. First, $\mathcal{A}$ checks locally in two phases if the input graph G is good. In the first phase, each node detects in five communication rounds whether it is contained in a valid gadget or neighboured by valid gadgets. If so, then in the second phase each node v corresponding to a node v' from the original directed graph checks that in the orientation obtained from the neighbouring gadgets, the node v' either has

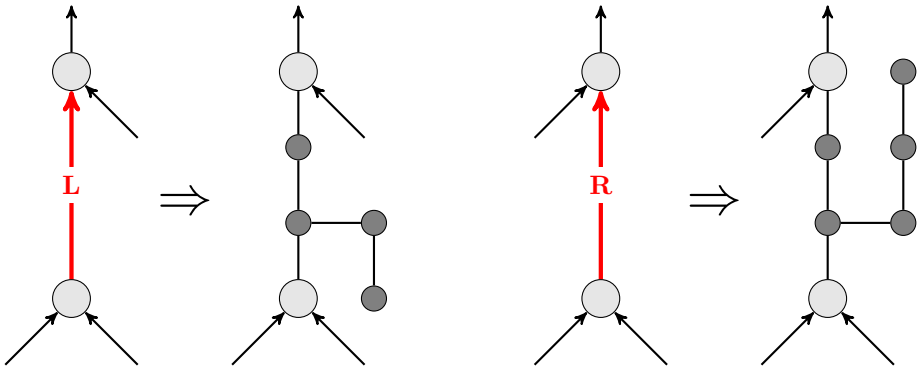


Figure 1 The red edge is replaced by the dark grey gadget consisting of four or five nodes, depending on the label. Similar encoding is performed to each edge of the original directed graph.

- two incoming edges (labelled "L" and "R") and one outgoing edge,
- only one outgoing edge (it is a leaf node), or
- only two incoming edges (it is the root node in case of a tree).

If either of the checks fails, the node broadcasts an instruction to halt to its neighbours and halts. Otherwise, it follows that the original graph G' is indeed a directed binary pseudotree with each edge directed away from the leaf nodes.

After verifying the goodness of the instance, algorithm $\mathcal{A}$ starts to count the distances to original leaf nodes. In the first round, each original leaf node broadcasts message 0 to its neighbours, halts and outputs 0. When a node that has not yet halted receives message i, where $i \in \{0, 1\}$, it broadcasts $i + 1 \bmod 2$ to its neighbours, halts and outputs $i + 1 \bmod 2$. Because the input graph is assumed to be finite and connected, each node eventually halts.

Since verifying the goodness takes only a constant number of rounds, and in the counting phase we count only up to 2, algorithm $\mathcal{A}$ needs only a constant number of states. Furthermore, as in any binary pseudotree the distance from any node to the closest leaf node is at most $O(\log n)$, each node halts after at most $O(\log n)$ rounds. On the other hand, in a balanced binary tree, the distance from the root to leaf nodes is $\Omega(\log n)$, and thus $\Omega(\log n)$ rounds are needed until all the nodes can output the parity of their distance to the closest original leaf. This completes the proof. ◀

4 Graphs of maximum degree 2

We are now ready to state the main theorem of this work.

▶ **Theorem 5.** *There exist a decision graph problem* Π *with constant space complexity and* $\Theta(n)$ *time complexity. Furthermore, the time complexity of* Π *is* $\Theta(n)$ *also in the class of graphs of maximum degree at most 2.*

To define our graph problem, we will make use of the Thue–Morse sequence via the following definition – compare this to Definition 3 earlier.

▶ **Definition 6** (Valid words)**.** Define a set of words over $\{0, 1, \_\}$ recursively as follows:

1. _0_ is *valid*,
2. if X is valid and Y is obtained from X by applying the substitutions $0 \mapsto$ 0_1_1_0 and $1 \mapsto$ 1_0_0_1 to each occurrence of 0 and 1, then Y is *valid*.

That is, valid words are obtained from the sequences $T_{2i}, i \in \mathbb{N}$, by inserting an underscore at the beginning, in between each symbol and at the end.

Now, we will define a decision graph problem that we call ThueMorse as follows.

▶ **Definition 7** (ThueMorse)**.** The local inputs of the problem are taken from the set $I = \{\alpha, \beta, \gamma\} \times \{0, 1, \_\}$ and each local output is either yes or no. Given an input function $f\colon V \to I$, we write $f = (f_1, f_2)$. Now, an instance (G, f) is a yes-instance of ThueMorse if and only if

- the graph $G = (V, E)$ is a path graph,
- for each $v \in V$, we have

$$\{f_1(u) : u = v \text{ or } u \in N(v)\} = \{\alpha, \beta, \gamma\},$$

  that is, by setting $\alpha < \beta < \gamma$, the first parts $f_1(\cdot)$ of the local inputs define a consistent orientation for the path,
- if we denote $V = \{v_1, v_2, \ldots, v_n\}$ where $\{v_i, v_{i+1}\} \in E$ for each $i \in \{1, 2, \ldots, n-1\}$, the word

$$f_2(v_1) f_2(v_2) \ldots f_2(v_n)$$

  defined by the second parts of the local inputs is valid.

4.1 Definition of the algorithm

Next, we will give an algorithm that is able to solve the decision problem ThueMorse by using only constant space. The high-level idea is as follows: First, we check that the local neighbourhood of each node looks correct. Then, we repeatedly apply substitutions that roll the configuration of the path graph back to a shorter prefix of the Thue–Morse sequence, until we reach a trivial configuration and accept the input – or fail to apply the substitutions unambiguously and consequently reject the input.

▶ **Lemma 8.** *There exists an algorithm $\mathcal{A}$ that solves problem* ThueMorse *in space $O(1)$ and time $O(n)$.*

First, let us introduce some terminology and notation. In an instance (G, f) of ThueMorse where G is either a path or a cycle graph, the sequence of symbols defined by f_2 in a given node neighbourhood is called the *input word* – note that contrary to usual words, the input word is unoriented. Sometimes we identify the node neighbourhood with the corresponding input word; the meaning should be clear from the context. During the execution of the algorithm, each node v has a *current symbol* $c(v)$ as part of its state. The vertical bar | shall denote the end of the path graph. We will make use of it by assuming that each degree-1 node v sees a "virtual" neighbour u with the symbol |, that is, $f_1(u) = f_2(u) = |$ and $c(u)$ is always equal to |. The underscore symbols _ will be called *separators*. Denote the alphabet which contains the possible current symbols by $\Sigma = \{0, 1, \_, |\}$.

We define the algorithm $\mathcal{A}$ in three parts, which we denote by I, II and III. In each part, any node can *abort*, which means that it sends a special abort message to its neighbours and then moves to state no and thus halts. If a node receives the abort message at any time, it aborts – that is, it passes the message on to all its neighbours, and then moves to state no and halts. At initialisation, each node v sets its symbol $c(v)$ to be equal to $f_2(v)$, the second part of the local input.

In **part I**, each node v verifies its degree and the orientation: if $\deg(v) \in \{1, 2\}$ and three different symbols from $\{\alpha, \beta, \gamma, |\}$ can be found in the local inputs within the radius-1

neighbourhood of v (that is, we have $|\{f_1(u) : u \in N(v) \cup \{v\}\}| = 3$), continue; otherwise, abort. Recall that the graph is assumed to be finite and connected. If none of the nodes aborts, it follows that the graph is either a path or a cycle and that the local inputs given by f_1 define a word of the form $\ldots\alpha\beta\gamma\alpha\beta\gamma\alpha\beta\gamma\alpha\ldots$.

In **part II**, each node v verifies the input word in its radius-1 neighbourhood: if the neighbourhood is in {|_0, 0_|, 0_0, 1_1, 0_1, 1_0, _0_, _1_}, continue; otherwise, abort. Note that here we do not care about the orientation of the word, and hence the set of symbols received from the neighbours is enough for this step. If none of the nodes aborts, the input word is *locally valid*: every other symbol is the separator _ and every other symbol is either 0 or 1.

Then, we proceed to **part III**, which contains the most interesting steps of the algorithm. Now we will make use of the orientation given as part of the input. Define $\alpha < \beta$, $\beta < \gamma$ and $\gamma < \alpha$. For each node v, we say that neighbour u of v is the *left neighbour* of v if $f_1(u) < f_1(v)$ and the *right neighbour* of v if $f_1(u) > f_1(v)$. Thus, each node can essentially send a different message to each of its two neighbours: the orientation symbol of the recipient indicates which part of the message is intended for which recipient. From now on, we will also assume that each node attaches its own orientation symbol as part of every message sent, so that nodes can distinguish between incoming messages.

Part III will consist of several *phases*. In each phase, we first gather information from the neighbourhood in two *buffers* and then try to apply a substitution to the word obtained in the buffers. More precisely, each node v has two buffers, the left buffer $L(v)$ and the right buffer $R(v)$ as part of its state. The buffers are used to store a *compressed* version of the input word in the neighbourhood: a sequence of consecutive symbols 0 or 1 is represented by a single 0 or 1, respectively.

Let us define some notation. Let $l, r\colon \Sigma^* \times \Sigma \to \Sigma^*$ be as follows. If $X = Ay$ for some word A, set $r(X, y) = X$. Otherwise, set $r(X, y) = Xy$. Function l is defined analogously: if $X = yA$ for some word A, set $l(X, y) = X$; otherwise, set $l(X, y) = yX$. In other words, functions l and r append a new symbol either to the beginning or to the end of a word, respectively – but only if the new symbol is different from the current first or last symbol of the word, respectively.

Now, in each phase, each node v first *fills its buffers* as follows. To initialise, node v broadcasts message $c(v)$ (its own current symbol) to its neighbours. Then, node v repeats the following. It sets the left buffer $L(v)$ equal to the message it receives from its left neighbour and the right buffer $R(v)$ equal to the message it receives from the right neighbour. After that, node v sends $r(L(v), c(v))$ to the right and $l(R(v), c(v))$ to the left. These steps are repeated until both buffers of v contain either 8 instances of the separator _ or an end-of-the-path marker |. When that happens, node v is finished with filling its buffers.

Next, node v combines its buffers to construct a *compressed view* of the input word in its neighbourhood. To that end, define $C\colon \Sigma^* \times \Sigma \times \Sigma^* \to \Sigma^*$ and $p\colon \Sigma^* \times \Sigma \times \Sigma^* \to \mathbb{N}$ as follows. If $X = Ay$ and $Z = yB$ for some $A, B \in \Sigma^*$, set $C(X, y, Z) = AyB$ and $p(X, y, Z) = |A|$. Otherwise, if $X = Ay$ for some $A \in \Sigma^*$, set $C(X, y, Z) = AyZ$ and $p(X, y, Z) = |A|$, and if $Z = yB$ for some $B \in \Sigma^*$, set $C(X, y, Z) = XyB$ and $p(X, y, Z) = |X|$. Else, set $C(X, y, Z) = XyZ$ and $p(X, y, Z) = |X|$. Now, the compressed view of node v is $V(v) = C(L(v), c(v), R(v))$, and $c(v)$ is at position $q(v) = p(L(v), c(v), R(v)) + 1$ in $V(v)$. In other words, the left buffer, the current symbol and the right buffer are concatenated in a way that removes successive repetitions of the same symbol.

Finally, node v does subword matching on the view $V(v)$. If $V(v)$ is equal to |_0_| or |_0_1_1_0_|, node v instructs other nodes to accept, moves to the yes state and halts.

Otherwise, node v searches $V(v)$ for the subword _0_1_1_0_1_0_0_1_ in all possible positions. Given a match, let the ith symbol of the subword be aligned with the $q(v)$th symbol of $V(v)$. If $i \in \{1, 9, 17\}$, set $c' = \_$. Otherwise, if $i \leq 8$, set $c' = 0$, and if $i \geq 10$, set $c' = 1$. After that, node v performs the same procedure with the reversed subword _1_0_0_1_0_1_1_0_ – but now, if $i \leq 8$, set $c' = 1$, and if $i \geq 10$, set $c' = 0$. If no matches could be found in $V(v)$, node v aborts. If several matches were found and they resulted in different values for c', node v aborts. Otherwise, node v updates its current symbol $c(v)$ to be the unambiguous value c'. This concludes the phase; if not aborted, on the next round, a new phase starts from the beginning. See Example 9 for illustrations of the matching and substitution steps in a few cases.

▶ **Example 9.** Consider the execution of part III of algorithm $\mathcal{A}$ in the following instances.

1. Path graph, yes-instance:

   |_0_1_1_0_1_0_0_1_1_0_0_1_0_1_1_0_|
   ⇓ (unambiguous substitutions)
   |_0000000_1111111_1111111_0000000_|
   ⇓ ($V(v) =$ |_0_1_1_0_|)
   accept

2. Path graph, no-instance:

   |_0_1_1_0_1_0_0_1_1_0_1_0_0_1_|
   ⇓
   |_0000000_1111111_ . . .
   . . . _0000000_1111111_|
   ⇓ (ambiguous substitutions)
   abort

3. Cycle graph (the ends marked with . . . are connected circularly):

   . . . _0_1_1_0_1_0_0_1_1_0_0_1_0_1_1_0_ . . .
   ⇓ (unambiguous substitutions)
   . . . _0000000_1111111_1111111_0000000_ . . .
   ⇓ (no matches)
   abort

4.2 Proof of correctness

In this section, we show that algorithm $\mathcal{A}$ executes correctly and always halts with the desired output. Since parts I and II are quite trivial, we will start with part III of the algorithm.

We say that a word $X = x_1 x_2 \dots x_i$ is the *compressed version* of a word $Y = y_1 y_2 \dots y_j$ if $x_i \neq x_{i+1}$ for all $i \in \{1, 2, \dots, i-1\}$ and there exist a surjective *compression mapping* $f \colon [j] \to [i]$ such that $f(1) = 1$, $f(k) \in \{f(k-1), f(k-1)+1\}$ for all $k \in \{2, 3, \dots, j\}$ and $y_k = x_{f(k)}$ for all $k \in [j]$.

▶ **Lemma 10.** *After any node v has finished collecting the buffers, $V(v)$ is the compressed version of the actual input word in the neighbourhood of v.*

Proof. We use induction on the number of rounds t after starting the phase. After the first round of the phase, each node v has received $L(v) = c(u)$ from its left neighbour u and $R(v) = c(w)$ from its right neighbour w. Hence $L(v)$ and $R(v)$ are compressed versions of the left and right 1-neighbourhoods of v, respectively.

Assume then that $L(u)$ and $R(w)$ are compressed versions of the left and right $(t-1)$-neighbourhoods of u and w, respectively, and let f_l and f_r be the corresponding mappings. Consider now the definition of the algorithm. The definition of the mapping r implies that what v receives from the left in round t of the phase is $L(u)$ – extended by $c(u)$ if and only if $c(u)$ differs from the last symbol of $L(u)$. If it does differ, we extend f_l by defining $f_l(t) = |L(u)| + 1$, otherwise $f_l(t) = |L(u)|$. Hence the new value of the buffer $L(v)$ is a compressed version of the left t-neighbourhood of v. The case of $R(v)$ is handled analogously.

Suppose that v has finished collecting the buffers. Now $L(v)$ and $R(v)$ are the compressed versions of the left and right k-neighbourhoods of v for some k. Then, it follows from the definition of the function C that an appropriate compression mapping can be formed and $V(v) = C(L(v), c(v), R(v))$ is the compressed version of the k-neighbourhood of v. ◀

We call the sequence of current symbols $c(u)$ in the graph a *configuration* (in the case of a cycle graph, the sequence is infinite in both directions). A maximal sequence of adjacent nodes such that each node v has the same current symbol $c(v)$ is called a *block*. We sometimes identify a block with the subword consisting of the current symbols of the block nodes.

In the next two lemmas, we show how updating the current symbol locally in nodes results in global substitutions in the configuration.

▶ **Lemma 11.** *Assume that in the current configuration, each maximal subword of the form* $0+$ *or* $1+$ *is of length* ℓ *and each maximal subword of the form* $\_+$ *is of length 1. If the algorithm is executed for one phase and no node aborts, in the resulting configuration the lengths are* $4\ell + 3$ *and 1, respectively. More precisely, the execution of one phase always results in substitutions of the following kinds:*

$$\_0^\ell\_1^\ell\_1^\ell\_0^\ell\_1^\ell\_0^\ell\_0^\ell\_1^\ell\_ \mapsto \_0^{4\ell+3}\_1^{4\ell+3}\_, \tag{1}$$

$$\_1^\ell\_0^\ell\_0^\ell\_1^\ell\_0^\ell\_1^\ell\_1^\ell\_0^\ell\_ \mapsto \_1^{4\ell+3}\_0^{4\ell+3}\_. \tag{2}$$

Proof. Since no node aborts, each node v is able to find an unambiguous new value for $c(v)$. Consider an arbitrary node u. Suppose that the pattern $P_1 = \_0\_1\_1\_0\_1\_0\_0\_1\_$ matches $V(u)$ so that the ith symbol of the pattern is aligned with the $q(v)$th symbol of $V(u)$. Now it follows from Lemma 10 that P_1 is the compressed version of an actual neighbourhood N of u. Due to the assumption, for each j such that the jth symbol of P_1 is 0 or 1, in the neighbourhood there are exactly ℓ consecutive symbols 0 or 1, respectively, that are mapped to the jth symbol of P_1 by the compression mapping f.

Let us consider the case $i = 6$ as an example. Now $c(u) = 1$. As the buffers are gathered until each of them contain eight separators _, the next 5 blocks to the left from the block of u, as well as the next 11 blocks to the right, gather views that are compressed versions of a neighbourhood containing N. Hence P_1 matches also their views. For example, for all nodes w in the block two steps left from the block of u, P_1 matches $V(w)$ at position $i - 2 = 4$. It follows from the definition of the algorithm that the new value for $c(u)$, as well as for $c(w)$, where w is in the next 4 blocks to the left from u or 2 blocks to the right from u, is 0. The new value for the node in the 5th block left from u as well as 3rd block right from u is _. Thus, after the phase, node u will be part of a 0-block of length $\ell + 1 + \ell + 1 + \ell + 1 + \ell = 4\ell + 3$.

The cases for all other values of i, as for as the matching the reverse pattern P_2, are analogous. ◀

We call a word $\_x_1^i\_x_2^i\_ \dots \_x_p^i\_$ a *padded Thue–Morse word of length* p if $x_1 x_2 \dots x_p$ is a prefix of the Thue–Morse sequence.

▶ **Lemma 12.** *Let W be a subword of a configuration C at the beginning of a phase. Let $k \geq 3$. If W is a (complement of a) padded Thue–Morse word of length 2^k and the algorithm is executed for one phase on C without aborting, the subword W is transformed to a (complement of a) padded Thue–Morse word of length 2^{k-2}.*

Proof. We use induction on k. If $k = 3$, we have $W = \_0^i\_1^i\_1^i\_0^i\_1^i\_0^i\_0^i\_1^i\_$ or $W = \_1^i\_0^i\_0^i\_1^i\_0^i\_1^i\_1^i\_0^i\_$ for some i. Then Lemma 11 implies that W is transformed to $\_0^j\_1^j\_$ or $\_1^j\_0^j\_$, respectively, where $j = 4i + 3$. Hence the claim holds for $k = 3$.

Suppose then that $k > 3$ and the claim holds for each subword that is a (complement of a) padded Thue–Morse word of length 2^{k-1}. Let W be a padded Thue–Morse word of length 2^k. Now the definition of the Thue–Morse sequence implies that we can write $W = W_1\_W_2$, where $W_1\_$ is a padded Thue–Morse word of length 2^{k-1} and $\_W_2$ is a complement of a padded Thue–Morse word of length 2^{k-1}. By the inductive hypothesis, $W_1\_$ is transformed to a padded Thue–Morse word of length 2^{k-3} and $\_W_2$ is transformed to a complement of a padded Thue–Morse word of length 2^{k-3}. Now the definition of the Thue–Morse sequence again implies that $W = W_1\_W_2$ is transformed to a padded Thue–Morse word of length 2^{k-2}. The case where W is a complement of a padded Thue–Morse word is completely analogous. ◀

Now we are ready to aggregate our previous lemmas to establish that algorithm $\mathcal{A}$ actually works correctly.

▶ **Lemma 13.** *In a yes-instance, each node eventually halts and outputs* yes.

Proof. Let (G, f) be a yes-instance of ThueMorse. By definition, algorithm $\mathcal{A}$ executes parts I and II successfully. Notice then that since the input word is valid, the configuration at the beginning of part III is actually a padded Thue–Morse word. If the input word is either _0_ or _0_1_1_0, the nodes move immediately to the yes state and halt. Otherwise, the input word is a padded Thue–Morse word of length at least 16, and by iterating Lemma 12 we obtain a shorter padded Thue–Morse word after every phase. Note that the new current symbol will be unambiguous for each node on each phase. Eventually the configuration will match with _0_1_1_0_, and the instance will get accepted. ◀

▶ **Lemma 14.** *In a no-instance, each node eventually halts and outputs* no.

Proof. Let (G, f) be a no-instance of ThueMorse. By assumption, G is finite and connected. If G contains a node of degree higher than 2, algorithm $\mathcal{A}$ aborts in part I. Hence we can assume that G has maximum degree at most two. It follows immediately from Lemma 11 that algorithm $\mathcal{A}$ halts on (G, f): since the size of blocks of 0's or 1's grows in each phase, we will eventually run out of nodes.

Graph G is either a path or a cycle graph. If $\mathcal{A}$ rejects in part I or part II, we are done. Hence, we can assume that the input f defines a consistent orientation and each 1-neighbourhood is of the correct form – that is, every second symbol on the input word is a separator _. Now it follows from Lemma 11 that the computation proceeds synchronously: in part III, each node starts a new phase in the same round.

Consider the case that G is a cycle graph. Due to Lemma 11, the number of blocks decreases after each phase. Eventually, if no node rejects, the execution reaches a configuration with $b \leq 4$ blocks of 0's and 1's. Then, each node sees the same sequence of b blocks repeating in its view. It follows that neither pattern P_1 or P_2 matches and the node rejects.

Assume then that G is a path graph. If there exists a yes-instance (G', f') with the same number of nodes as G has, we proceed as follows. Suppose for a contradiction that (G, f) gets accepted. Then there is a smallest i such that before the ith phase, the configurations C on (G, f) and C' on (G', f'), respectively, are different, but after the ith phase, they are identical, C''. It follows that C'' contains a subword of the form $\_0^i\_1^i\_$ for some i, such that C and C' differ on a position overlapping with the subword. But this is a contradiction, as it follows from Lemma 11 that C and C' cannot differ on such a position.

Finally, consider the case where (G, f) is a no-instance such that there does not exist a yes-instance with the same number of nodes. Suppose again for a contradiction that (G, f) gets accepted. Let (G', f') be the largest yes-instance no larger than (G, f) and let (G'', f'') be the yes-instance that is one step larger from (G', f'). Lemma 12 implies that the algorithm $\mathcal{A}$ needs exactly one phase more on (G'', f'') than on (G', f'). The number of phases on (G, f) equals either the number of phases on (G', f') or on (G'', f''). But either case is a contradiction, since the instances have a different amount of blocks in the beginning, and Lemma 11 implies that the size of blocks grows at a fixed rate. ◀

The last thing left to do is analysing the complexity of the algorithm. This is taken care of by the following lemmas.

▶ **Lemma 15.** *The space usage of algorithm $\mathcal{A}$ is in $O(1)$.*

Proof. Parts I and II of algorithm $\mathcal{A}$ clearly use only a constant number of states. Consider then part III. When gathering the buffers, consecutive blocks of symbols in the neighbourhood are represented by only one symbol, and only a constant amount of blocks are gathered (eight separator symbols _ in each buffer). Hence a constant number of states is enough to represent the contents of the buffers. Furthermore, the buffers get erased after each phase has completed. Thus, algorithm $\mathcal{A}$ can be implemented using a constant number of states, independent of the size of the input. ◀

▶ **Lemma 16.** *The running time of algorithm $\mathcal{A}$ is in $O(n)$.*

Proof. Executing each phase of part III of the algorithm can be done in $8i + 8 = 8(i + 1)$ rounds, where i is the size of the blocks at the start of the phase. Recall that after each phase, the size of the blocks grows from ℓ to $4\ell + 3$. Note also that $\frac{1}{2} \log n$ phases are enough to grow the block size so large that the algorithm has to either accept or reject. It follows that

$$8(1 + 1) + 8(7 + 1) + 8(31 + 1) + \cdots + 8(2^{2(\frac{1}{2} \log n)-1)}) \leq 8n$$

is an upper bound for the running time. ◀

On the other hand, $\Omega(n)$ rounds are clearly necessary to solve ThueMorse: the node at the end of the path has to receive information from the other end of the path to be able to verify that the instance is actually a yes-instance. This concludes the proof of Theorem 5.

5 Conclusions

We introduced space complexity as a new dimension in the classification of distributed graph problems. In particular, we identified a model of computation and a class of graphs, where the question on the existence of constant-space, non-constant-time algorithms is interesting and non-trivial. The answer turned out to be positive.

The result opens up the way to study constant space further – we can ask, for example, what other time complexities besides $\Theta(\log n)$ and $\Theta(n)$ can possibly be found and in which graph classes. Another open question of interest is the existence of natural graph problems that exhibit the constant-space non-constant-time characteristic.

Acknowledgements. We have discussed this work with numerous people, including at least Juho Hirvonen, Janne H. Korhonen, Antti Kuusisto, Joel Rybicki and Przemysław Uznański. Many thanks to all of them, as well as to others that we may have forgotten. We also thank the anonymous reviewers for their helpful feedback.

References

1 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. In *Proc. 25th International Symposium on Distributed Computing (DISC 2011)*, volume 6950 of *Lecture Notes in Computer Science*, pages 32–50. Springer, 2011. doi:10.1007/978-3-642-24100-0_3.

2 Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980)*, pages 82–93. ACM, 1980. doi:10.1145/800141.804655.

3 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity, 2017. arXiv:1711.01871.

4 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th Annual ACM Symposium on Theory of Computing (STOC 2016)*, pages 479–488. ACM, 2016. doi:10.1145/2897518.2897570. arXiv:1511.00900.

5 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 36th Annual ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110. ACM, 2017. doi:10.1145/3087801.3087833. arXiv:1702.05456.

6 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624. IEEE, 2016. doi:10.1109/FOCS.2016.72. arXiv:1602.08166.

7 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. In *Proc. 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2017)*, pages 156–167. IEEE, 2017. doi:10.1109/FOCS.2017.23. arXiv:1704.06297.

8 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an infinite space with finite memory scouts. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, pages 207–224. SIAM, 2017. doi:10.1137/1.9781611974782.14. arXiv:1704.02380.

9 Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proc. 24th International Symposium on Distributed Computing (DISC 2010)*, volume 6343 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2010. doi:10.1007/978-3-642-15763-9_15.

10 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, pages 471–482. Springer, 2014. doi:10.1007/978-3-662-43951-7_40. arXiv:1311.3062.

11 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proc. 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC 2013)*, pages 137–146. ACM, 2013. doi:10.1145/2484239.2484244. arXiv:1202.1186.

12 Martin Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

13 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proc. 49th Annual ACM Symposium on Theory of Computing (STOC 2017)*, pages 784–797. ACM, 2017. doi:10.1145/3055399.3055471. arXiv:1611.02663.

14 Mika Göös, Juho Hirvonen, and Jukka Suomela. Lower bounds for local approximation. *Journal of the ACM*, 60(5):39:1–23, 2013. doi:10.1145/2528405. arXiv:1201.6675.

15 Lauri Hella, Matti Järvisalo, Antti Kuusisto, Juhana Laurinharju, Tuomo Lempiäinen, Kerkko Luosto, Jukka Suomela, and Jonni Virtema. Weak models of distributed computing, with connections to modal logic. *Distributed Computing*, 28(1):31–53, 2015. doi:10.1007/s00446-013-0202-3. arXiv:1205.2051.

16 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: lower and upper bounds. *Journal of the ACM*, 63(2):17:1–17:44, 2016. doi:10.1145/2742012. arXiv:1011.5470.

17 Antti Kuusisto. Infinite networks, halting and local algorithms. In *Proc. 5th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2014)*, volume 161 of *Electronic Proceedings in Theoretical Computer Science*, pages 147–160, 2014. doi:10.4204/EPTCS.161.14. arXiv:1408.5963.

18 Antti Kuusisto and Fabian Reiter. Emptiness problems for distributed automata. In *Proc. 8th International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2017)*, volume 256 of *Electronic Proceedings in Theoretical Computer Science*, pages 210–222, 2017. doi:10.4204/EPTCS.256.15. arXiv:1705.02609.

19 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

20 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

21 Fabian Reiter. Distributed graph automata. In *Proc. 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2015)*, pages 192–201. IEEE, 2015. doi:10.1109/LICS.2015.27. arXiv:1404.6503.

22 Fabian Reiter. Asynchronous distributed automata: a characterization of the modal mu-fragment. In *Proc. 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, pages 100:1–100:14. Schloss Dagstuhl – Leibniz Center for Informatics, 2017. doi:10.4230/LIPIcs.ICALP.2017.100. arXiv:1611.08554.

23 John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

24 Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

Shape Formation by Programmable Particles*

Giuseppe A. Di Luna[1], Paola Flocchini[2], Nicola Santoro[3], Giovanni Viglietta[4], and Yukiko Yamauchi[5]

1 **University of Ottawa, Ottawa, Canada**
gdiluna@uottawa.ca
2 **University of Ottawa, Ottawa, Canada**
paola.flocchini@uottawa.ca
3 **Carleton University, Ottawa, Canada**
santoro@scs.carleton.ca
4 **University of Ottawa, Ottawa, Canada**
gvigliet@uottawa.ca
5 **Kyushu University, Fukuoka, Japan**
yamauchi@inf.kyushu-u.ac.jp

Abstract

Shape formation (or pattern formation) is a basic distributed problem for systems of computational mobile entities. Intensively studied for systems of autonomous mobile robots, it has recently been investigated in the realm of programmable matter, where entities are assumed to be small and with severely limited capabilities. Namely, it has been studied in the geometric Amoebot model, where the anonymous entities, called particles, operate on a hexagonal tessellation of the plane and have limited computational power (they have constant memory), strictly local interaction and communication capabilities (only with particles in neighboring nodes of the grid), and limited motorial capabilities (from a grid node to an empty neighboring node); their activation is controlled by an adversarial scheduler. Recent investigations have shown how, starting from a well-structured configuration in which the particles form a (not necessarily complete) triangle, the particles can form a large class of shapes. This result has been established under several assumptions: agreement on the clockwise direction (i.e., chirality), a sequential activation schedule, and randomization (i.e., particles can flip coins to elect a leader).

In this paper we provide a characterization of which shapes can be formed deterministically starting from any simply connected initial configuration of n particles. The characterization is constructive: we provide a universal shape formation algorithm that, for each feasible pair of shapes (S_0, S_F), allows the particles to form the final shape S_F (given in input) starting from the initial shape S_0, unknown to the particles. The final configuration will be an appropriate scaled-up copy of S_F depending on n.

If randomization is allowed, then any input shape can be formed from any initial (simply connected) shape by our algorithm, provided that there are enough particles.

Our algorithm works without chirality, proving that chirality is computationally irrelevant for shape formation. Furthermore, it works under a strong adversarial scheduler, not necessarily sequential.

We also consider the complexity of shape formation both in terms of the number of rounds and the total number of moves performed by the particles executing a universal shape formation algorithm. We prove that our solution has a complexity of $O(n^2)$ rounds and moves: this number of moves is also asymptotically worst-case optimal.

1998 ACM Subject Classification F.1.1 Models of Computation, F.2.2 Nonnumerical Algorithms and Problems, I.2.11 Distributed Artificial Intelligence, I.2.9 Robotics

* Full version available at https://arxiv.org/abs/1705.03538. A short version appeared as "Brief Announcement: Shape Formation by Programmable Particles" in Proceedings of DISC 2017.

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão; Article No. 31; pp. 31:1–31:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Shape formation, pattern formation, programmable matter, Amoebots, leader election, distributed algorithms

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.31

1 Introduction

1.1 Background

The term *programmable matter*, introduced by Toffoli and Margolus over a quarter century ago [23], is used to denote matter that has the ability to change its physical properties (e.g., shape, color, etc.) in a programmable fashion, based upon user input or autonomous sensing. Often programmable matter is envisioned as a very large number of very small locally interacting computational particles, programmed to collectively perform a complex task. Such particles could have applications in a variety of important situations: smart materials, autonomous monitoring and repair, minimal invasive surgery, etc.

Several theoretical models for programmable matter have been proposed, ranging from DNA self-assembly systems (e.g., [18, 19, 21]) to metamorphic robots (e.g., [3, 16, 24]) to nature-inspired synthetic insects and micro-organisms (e.g., [10, 11]).

Of particular interest, from the distributed computing viewpoint, is the *geometric Amoebot* model [2, 5, 6, 7, 8, 9, 10, 14]. In this model, introduced in [10] and so called because inspired by the behavior of amoeba, programmable matter is viewed as a swarm of decentralized autonomous self-organizing entities, operating on a hexagonal tessellation of the plane. These entities, called *particles*, are constrained by having simple computational capabilities (they are finite-state machines), strictly local interaction and communication capabilities (only with particles located in neighboring nodes of the hexagonal grid), and limited motorial capabilities (from a grid node to an empty neighboring node); furthermore, their activation is controlled by an adversarial (but fair) synchronous scheduler. A feature of the Amoebot model is that particles can be in two modes: *contracted* and *expanded*. When contracted, a particle occupies only one node, while when expanded the particle occupies two neighboring nodes; it is indeed this ability of a particle to expand and contract that allows it to move on the grid.

In the Amoebot model, the research focus has been on applications such as coating [5, 9], gathering [2], and shape formation [7, 8, 10]. The latter is also the topic of our investigation.

The *shape formation* problem is prototypical for systems of self-organizing entities. This problem, called *pattern formation* in swarm robotics, requires the entities to move in the spatial universe they inhabit in such a way that, within finite time, their positions form the geometric shape given in input (modulo translation, rotation, scaling, and reflection), and no further changes occur. Indeed, this problem has been intensively studied in active systems such as autonomous mobile robots (e.g., [1, 4, 12, 13, 22]) and modular robotic systems (e.g., [17, 20]).

Shape formation for Amoebots has been investigated in [7, 8, 10], taking into account that, due to the ability of particles to expand, it might be possible to form shapes whose size is larger than the number of particles.

The pioneering study of [7] on *shape formation* in the geometric Amoebot model showed how particles can build simple shapes, such as a hexagon or a triangle. Subsequent investigations [8] have recently shown how, starting from a well-structured configuration where the particles form a (not necessarily complete) triangle, they can form a larger class of shapes under several assumptions, including randomization (which is used to elect a leader), chirality

(i.e., a globally consistent circular orientation of the plane shared by all particles), and a sequential activation schedule (i.e., at each time unit the scheduler selects only one particle which will interact with its neighbors and possibly move).

These results and assumptions immediately and naturally open fundamental research questions, including: Are other shapes formable? What can be done deterministically? Is chirality necessary? What happens if the scheduler is not sequential? What if the initial configuration is not well structured? Notice that, without the availability of a unique leader (provided by randomization), dropping the chirality assumption becomes a problem with a non-sequential schedule.

In this paper, motivated and stimulated by these questions, we continue the investigation on shape formation in the geometric Amoebot model and provide some definitive answers.

1.2 Main Contributions

We continue the investigation, significantly extending the existing results. Among other things, we provide a constructive characterization of which shapes S_F can be formed *deterministically* starting from an unknown *simply connected* initial configuration S_0 of n particles (i.e., a connected configuration without "holes").

As in [8], we assume that the size of the description of S_F is constant with respect to the size of the system, so that it can be encoded by each particle as part of its internal memory. Such a description is available to all the particles at the beginning of the execution, and we call it their "input". The particles will form a final configuration that is an appropriate scaling, translation, rotation, and perhaps reflection of the input shape S_F. Since all particles of S_0 must be used to construct S_F, the scale λ of the final configuration depends on n: we stress that λ is unknown to particles, and they must determine it autonomously.

Given two shapes S_0 and S_F, we say that the pair (S_0, S_F) is *feasible* if there exists a deterministic algorithm that, in every execution and regardless of the activation schedule, allows the particles to form S_F starting from S_0 and no longer move.

On the contrary, a pair (S_0, S_F) of shapes is *unfeasible* when the symmetry of the initial configuration S_0 prevents the formation of the final shape S_F. In Section 2, we formalize the notion of *unbreakable* symmetry of shapes embedded in triangular grids, and in Theorem 1 we show that starting from an unbreakable k-symmetric configuration only unbreakable k-symmetric shapes can be formed.

For all the feasible pairs, we provide a *universal shape formation algorithm* in Section 3. This algorithm does not need any information on S_0, except that it is simply connected.

These results concern the *deterministic* formation of shapes. As a matter of fact, our algorithm uses a deterministic leader election algorithm as a subroutine (Sections 3.1–3.4). If the initial shape S_0 is unbreakably k-symmetric, such an algorithm may elect as many as k neighboring leader particles, where $k \in \{1, 2, 3\}$. It is trivial to see that, with a constant number of coin tosses, we can elect a unique leader among these k with arbitrarily high probability. Thus, our results immediately imply the existence of a *randomized* universal shape formation algorithm for *any* pair of shapes (S_0, S_F) where S_0 is simply connected. This extends the result of [8], which assumes the initial configuration to be a (possibly incomplete) triangle.

Additionally, our notion of shape generalizes the one used in [8], where a shape is only a collection of triangles, while we include also 1-dimensional segments as its constituting elements. In Section 4 we will show how the concept of shape can be further generalized to include essentially anything that is Turing-computable.

Our algorithm works under a stronger adversarial scheduler that activates an arbitrary number of particles at each stage (i.e., not necessarily just one, like the sequential scheduler), and with a slightly less demanding communication system.

Moreover, in our algorithm no chirality is assumed: indeed, unlike in [8], different particles may have different handedness. On the contrary, in the examples of unfeasibility given in Theorem 1, all particles have the same handedness. Together, these two facts allows us to conclude that *chirality is computationally irrelevant* for shape formation.

Finally, we analyze the complexity of shape formation in terms of the total number of *moves* (i.e., contractions and expansions) performed by n particles executing a universal shape formation algorithm, as well as in terms of the total number of *rounds* (i.e., spans of time in which each particle is activated at least once, also called *epochs*) taken by the particles. We first prove that any universal shape formation algorithm requires $\Omega(n^2)$ moves in some cases (Theorem 2). We then show that the total number of moves of our algorithm is $O(n^2)$ in all cases (Theorem 3): that is, our solution is asymptotically worst-case optimal. The time complexity of our algorithm is also $O(n^2)$ rounds, and optimizing it is left as an open problem (we can reduce it to $O(n \log n)$, and we have a lower bound of $\Omega(n)$).

Obviously, we must assume the size of S_0 (i.e., the number of particles that constitute it) to be sufficiently large with respect to the input description of the final shape S_F. More precisely, denoting the size of S_F as m, we assume n to be lower-bounded by a cubic function of m. A similar restriction is also found in [8].

To the best of our knowledge, all the techniques employed by our universal shape formation algorithm are new.

Further technical details and most proofs can be found in the full version of the paper [15].

2 Model and Preliminaries

Particles. A *particle* is a conceptual model for a computational entity that lives in an infinite regular triangular grid G, which we imagine as embedded in the Euclidean plane $\mathbb{R}^2$. A particle may occupy either one vertex of G or two adjacent vertices: in the first case, the particle is said to be *contracted*; otherwise, it is *expanded*. When it is expanded, one of the vertices it occupies is called its *head*, and the other vertex is its *tail*. A particle may move through G by repeatedly expanding toward a neighboring vertex of G and contracting into its head.[1]

No vertex of G can ever be occupied by more than one particle at the same time. Accordingly, a contracted particle cannot expand toward a vertex that is already occupied by another particle. If two or more particles attempt to expand toward the same (unoccupied) vertex at the same time, only one of them succeeds, chosen arbitrarily by an adversarial scheduler (see below).

In our model, time is "discrete", i.e., it is an infinite ordered sequence of instants, called *stages*. Say that in the graph G there is a set P of particles, which we call a *system*. At each stage, some particles of P are *active*, and the others are *inactive*. We may think of the activation of a particle as an act of an adversarial *scheduler*, which arbitrarily and unpredictably decides which particles are active at each stage. The only restriction on the scheduler is a bland *fairness* constraint, requiring that each particle be active for infinitely many stages in total. That is, the scheduler can never keep a particle inactive forever.

[1] The model in [8] allows a special type of coordinated move called "handover". Since we will not need our particles to perform this type of move, we omit it from our model.

When a particle is activated for a certain stage, it "looks" at the vertices of G adjacent to its head, discovering if they are currently unoccupied, or if they are head or tail vertices of some particle. All particles are indistinguishable (i.e., they are *anonymous*). Each active particle may then decide to either expand, contract, or stay still for that stage. All these operations are performed by all active particles simultaneously, and take exactly one stage. So, when the next stage starts, a new set of active particles is selected, which observe their surroundings and move, and so on.

Each particle has an *internal state* that it can modify every time it is activated. The internal state of any particle must be picked from a finite set: particles have an amount of "memory" that is constant with respect to the size of the system, n.

Two particles can also *communicate* by sending each other *messages* taken from a finite set, provided that their heads are adjacent vertices of G. A particle reads the incoming messages from all its neighbors as soon as it is activated. If a second message is sent through an edge of G before the first one is read, the first message is overwritten and becomes inaccessible.

Each particle labels the six edges incident to each vertex of G with *port numbers*, going from 0 to 5. Each particle uses a consistent numbering that is invariant under translation on G. However, different particles may disagree on which of the edges incident to a vertex has port number 0 and whether the numbering should follow the clockwise or counterclockwise order: this is called the particles' *handedness*. So, the handedness of a particle does not change as the particle moves, but different particles may have different handedness.

At each stage, each active particle looks at its surroundings to see which neighboring vertices are occupied, and it reads the incoming messages. Based on these and on its internal state, the particle executes a *deterministic algorithm* that computes a new internal state, the messages to be sent to the neighbors, and whether the particle should expand to some adjacent vertex, contract, or stay still.

We assume that, when stage 0 starts, all particles are contracted, they all have the same predefined internal state, and there are no messages pending between particles.

Shape formation. A *shape* is a non-empty connected set consisting of the union of finitely many edges and faces of G. We stress that a shape is not a subgraph of an abstract graph, but it is a subset of $\mathbb{R}^2$, i.e., a geometric set. A shape S is *simply connected* if the set $\mathbb{R}^2 \setminus S$ is connected (intuitively, S has no "holes"). The *size* of a shape is the number of vertices of G that lie in it.

We say that two shapes S and S' are *equivalent* if S' is obtained from S by a similarity transformation, i.e., a composition of a translation, a rotation, an isotropic scaling by a positive factor, and an optional reflection. A shape S is *minimal* if no shape that is equivalent to it has a smaller size. Let σ be a similarity transformation such that $S' = \sigma(S)$, where S is minimal: we say that the (positive) scale factor of σ is the *scale* of S'.

A system of particles in G *forms* a shape S if the vertices of G that are occupied by particles are exactly the ones that lie in S. An (S_0, S_F)*-shape formation algorithm* is an algorithm that makes a system of particles form a shape *equivalent* to S_F (not necessarily S_F itself), provided that they form shape S_0 at stage 0. That is, however the port labels of each particle are arranged, and whatever the choices of the scheduler are. After forming the shape, the particles should no longer move. If such an algorithm exists, (S_0, S_F) is called a *feasible* pair of shapes.

In the rest of this paper, we will characterize the feasible pairs of shapes (S_0, S_F), provided that S_0 is simply connected and its size is not too small. That is, for every such pair of

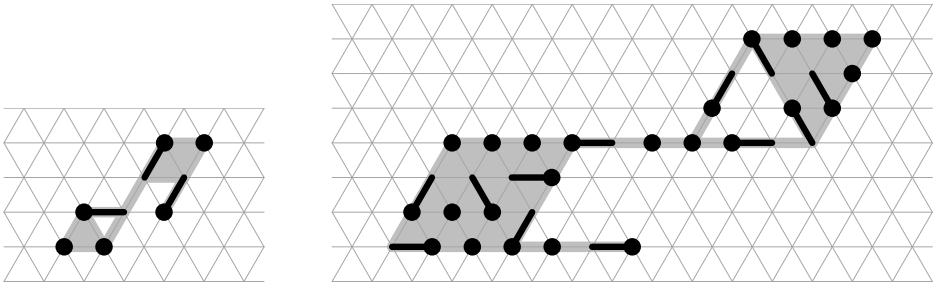


Figure 1 Two systems of particles forming equivalent shapes. Contracted particles are represented as black dots; expanded particles are black segments (a dot represents a particle's head). Shapes are indicated by gray blobs. The shape on the left is minimal; the one on the right has scale 3.

shapes, we will either prove that no shape formation algorithm exists (Theorem 1), or we will give an explicit shape formation algorithm. We will actually give a *universal shape formation algorithm* (Theorem 3), which only takes the "final shape" S_F (or a representation thereof) as a parameter, and has no information on the "initial shape" S_0, except that it is simply connected. As in [8], we assume that the size of the parameter S_F is constant with respect to the size of the system, so that S_F can be encoded by each particle as part of its internal memory.

Unformable shapes. A shape is said to be *unbreakably k-symmetric*, for some integer $k > 1$, if it has a center of k-fold rotational symmetry that does not coincide with any vertex of G. Observe that there exist unbreakably k-symmetric shapes only for $k = 2$ and $k = 3$.

If the system initially forms an unbreakably k-symmetric shape, the port labels of symmetric particles happen to be symmetric, and the scheduler always activates symmetric particles simultaneously, then the system never ceases to form an unbreakably k-symmetric shape. Therefore, we have the following:

▶ **Theorem 1.** *If (S_0, S_F) is a feasible pair and S_0 is unbreakably k-symmetric, then any minimal shape that is equivalent to S_F is also unbreakably k-symmetric.*

In Section 3, we are going to prove that the condition of Theorem 1 characterizes the feasible pairs of shapes, provided that S_0 is simply connected, and the size of S_0 is large enough with respect to the size of S_F.

Measuring movements and rounds. We are also concerned to measure the total number of *moves* performed by a system of size n executing a shape formation algorithm. For instance, if the particles initially form a (full) regular hexagon, they must make at least $\Omega(n^2)$ moves in total to form a line segment (because $\Omega(n)$ particles must move over a distance of $\Omega(n)$). Hence we have the following lower bound:

▶ **Theorem 2.** *A system of n particles executing any universal shape formation algorithm performs $\Omega(n^2)$ moves in total.*

In Section 3, we will prove that our universal shape formation algorithm requires $O(n^2)$ moves in total, and is therefore worst-case optimal with respect to this parameter.

Similarly, we want to measure how many *rounds* it takes the system to form the final shape (a round is a span of time in which each particle is activated at least once). We will show that our universal shape formation algorithm takes $O(n^2)$ rounds.

3 Universal Shape Formation Algorithm

Algorithm structure. The universal shape formation algorithm takes a "final shape" S_F as a parameter: this is encoded in the initial states of all particles. Without loss of generality, we will assume S_F to be minimal. The algorithm consists of seven phases:

1. A *lattice consumption phase*, in which the initial shape S_0 is "eroded" until 1, 2, or 3 pairwise adjacent particles are identified as "candidate leaders". No particle moves in this phase: only messages are exchanged. This phase ends in $O(n)$ rounds.
2. A *spanning forest construction phase*, in which a spanning forest of S_0 is constructed, where each candidate leader is the root of a tree. No particle moves, and the phase ends in $O(n)$ rounds.
3. A *handedness agreement phase*, in which all particles assume the same handedness as the candidate leaders (some candidate leaders may be eliminated in the process). In this phase, at most $O(n)$ moves are made. However, at the end, the system forms S_0 again. This phase ends in $O(n)$ rounds.
4. A *leader election phase*, in which the candidate leaders attempt to break symmetries and elect a unique leader. If they fail to do so, and $k > 1$ leaders are left at the end of this phase, it means that S_0 is unbreakably k-symmetric, and therefore the "final shape" S_F must also be unbreakably k-symmetric (cf. Theorem 1). No particle moves, and the phase ends in $O(n^2)$ rounds.
5. A *straightening phase*, in which each leader coordinates a group of particles in the formation of a straight line. The k resulting lines have the same length. At most $O(n^2)$ moves are made, and the phase ends in $O(n^2)$ rounds.
6. A *role assignment phase*, in which the particles determine the scale of the shape S'_F (equivalent to S_F) that they are actually going to form. Each particle is assigned an identifier that will determine its behavior during the formation process. No particle moves, and the phase ends in $O(n^2)$ rounds.
7. A *shape composition phase*, in which each straight line of particles, guided by a leader, is reconfigured to form an equal portion of S'_F. At most $O(n^2)$ moves are made, and the phase ends in $O(n^2)$ rounds.

No a-priori knowledge of S_0 is needed to execute this algorithm (S_0 just has to be simply connected), while S_F must of course be known to the particles and have constant size, so that its description can reside in their memory. Note that the knowledge of S_F is needed only in the last two phases of the algorithm.

Synchronization. As long as there is a unique (candidate) leader in the system, there are no synchronization problems: this one particle coordinates all others, and autonomously decides when each phase ends and the next phase starts.

However, if there are $k > 1$ (candidate) leaders, there are possible issues arising from the intrinsic asynchronicity of our particle model. Typically, a (candidate) leader will be in charge of coordinating only a portion of the system, and we want to avoid the undesirable situation in which different leaders are executing different phases of the algorithm.

So, a basic synchronization protocol is executed "in parallel" with the shape formation algorithm. This protocol simply makes sure that, whenever the (candidate) leaders are neighbors (which is true most of the time), they exchange messages containing the identifiers of the phases that they are currently executing, and those who are ahead wait until the others have caught up (see [15] for more details).

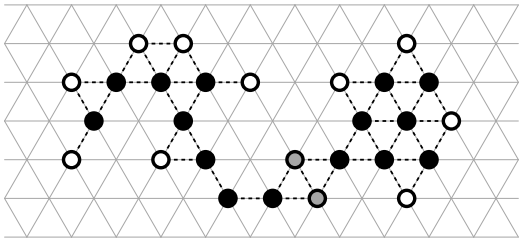


Figure 2 The particles in white or gray are corner particles; the two in gray are locked particles. Dashed lines indicate adjacencies between particles.

3.1 Lattice Consumption Phase

The goal of this phase is to identify 1, 2, or 3 *candidate leaders*. This is done without making any movements, but only exchanging messages. Each particle's internal state has a flag (i.e., a bit) called *Eligible*. All particles start the execution in the same state, with the Eligible flag set. As the execution proceeds, eligible particles will gradually "eliminate themselves" by clearing their Eligible flag. This is achieved through a process similar to erosion, which starts from the boundary of the initial shape and proceeds toward its interior.

The "consumption" of the initial shape S_0 starts from its *corner particles*: roughly speaking, these are the particles located at convex vertices of S_0 or at the end of dangling edges, as Figure 2 shows.

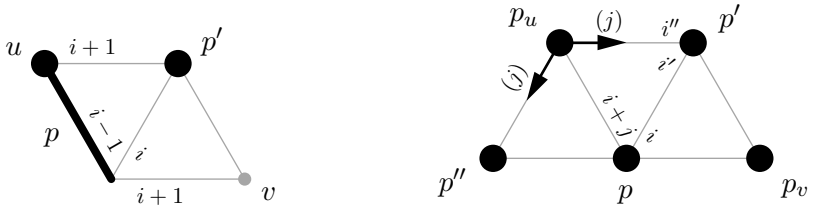
By looking at its surroundings, a particle knows if it is a corner particle or not; then, it can communicate this information to its neighbors. In most cases, if S_0 is a simply connected shape, removing any subset of its corner particles leaves the shape simply connected (i.e., it does not disconnect it or create holes in it). There is just one exception, represented by the two gray particles in Figure 2: removing both particles disconnects the shape. Fortunately, such a configuration is unique and can be locally recognized by the gray particles (provided that they send each other a description of their neighborhoods), which identify themselves as *locked particles*. A locked particle waits and remains eligible until some of its neighbors have eliminated themselves.

We can also prove that any simply connected shape contains non-locked corner particles, and therefore the erosion process eventually succeeds: when only 1, 2, or 3 pairwise adjacent eligible particles remain, they are the candidate leaders (technically, they become candidate leaders by setting an internal *Candidate* flag). Note that taking another erosion step from there may eliminate all particles (since they may behave symmetrically), and therefore we must stop at this point and use other methods to elect a leader.

3.2 Spanning Forest Construction Phase

The spanning forest construction phase starts when 1, 2, or 3 pairwise adjacent candidate leaders have been identified, and no other particle is eligible. In this phase, each candidate leader becomes the root of a tree embedded in G. Eventually, the set of these trees will be a spanning forest of the subgraph of G induced by the system of particles.

The algorithm is straightforward: the trees are constructed starting from the candidate leaders, and the process involves more and more adjacent particles until all of them are included. Each particle remembers which port label corresponds to its parent and which ones correspond to its children. The details of this algorithm can be found in [15].



(a) A neighbor u is unoccupied. **(b)** Both neighbors u and v are occupied.

Figure 3 The subroutine by which p communicates its handedness to p'. The labels on edges indicate port numbers. The labels in parentheses are messages.

3.3 Handedness Agreement Phase

When a candidate leader is notified by all its children that its tree can no longer be expanded, it transitions to the handedness agreement phase. Recall that each particle may label ports in clockwise or counterclockwise order: this is called the particle's handedness. By the end of this phase, all particles will agree on a common handedness. The agreement process starts at the candidate leaders and proceeds through the spanning forest constructed in the previous phase, from parents to children.

The core subroutine of this algorithm involves a generic particle p that intends to communicate its handedness to one of its children p' (in the tree constructed in the previous phase). Of course, this cannot be done simply by message passing, because a particle has no direct way to tell a neighbor which direction it "thinks" is clockwise: a special technique is required, which is summarized in Figure 3.

Let u and v be the two vertices of G that are neighbors to both p and p'. Suppose first that one of them is unoccupied, say u. Without loss of generality, p sees u to the left of p', as in Figure 3a. Then, p expands toward u and sends a message to p' saying "I-Moved-Left". If p' gets this message from its right neighbor, it has the same handedness as p; otherwise, it inverts its own handedness to match p's. Then, p goes back to its original location.

Suppose now that both u and v are occupied by particles p_u and p_v, as in Figure 3b. Then, p sends a message to p_u saying "You-are-my-Left-Neighbor" and one to p_v saying "You-are-my-Right-Neighbor". Then, p_u is supposed to message p', but it does not know where it is, except that it neighbors both p_u and p. As there are two such locations, p_u sends an "I-am-the-Left-Neighbor" message to both. Again, if p' receives this message from its right neighbor (with respect to its parent p), it knows it has the same handedness as p; otherwise, it inverts its handedness. In the meantime, p_v does a similar thing, sending "I-am-the-Right-Neighbor" messages. p' waits until it has received messages from both p_u and p_v, and then it notifies its parent p that the procedure is over.

This subroutine is executed between any non-leaf particle and all its children, starting from the candidate leaders. Before that, the candidate leaders execute a variation of this subroutine among themselves, to make sure they all have the same handedness (if they do not, then all but one are eliminated). At the end of this process, all particles have agreed on the same handedness.

The above algorithm has some obvious flaws, because several particles may be executing the subroutine at the same time, possibly interfering with each other if they try to expand toward the same vertex or if they send messages to the same particle. To solve the first problem, the particle p that initiates the subroutine does not try to expand to u if it

remembers that it was originally occupied by some other particle; instead, it waits for p_u to come back or expands to v. Also, if two particles try to expand toward the same location, only one succeeds: the other particle waits and tries again later.

To solve the second problem, when both u and v are occupied, p first *locks* both p_u and p_v, then notifies p', and then proceeds with the subroutine. When a particle is locked, it finishes the current subroutine before starting a new one with another neighbor. So, when p_u sends its "I-am-the-Left-Neighbor" or "I-am-the-Right-Neighbor" to the wrong particle p'', this particle is able to realize that it is not the intended addressee, and ignores the message. This is because the parent of p'' cannot have locked p_u and then notified p'', since p_u has already been locked by p for an operation with p'.

A consequence of this protocol is that, if p can lock only p_u, say, because p_v has already been locked by some other particle, then p keeps p_u locked while it waits for p_v to become unlocked: note that this potentially gives rise to *deadlocks*. To prevent them, each particle executes the subroutine with only one child at a time; when all its children have the same handedness, then it "authorizes" its first child to proceed with its own children, etc. This way, at any time there can be at most one pair of particles involved in the subroutine in each tree of the spanning forest. Since there are at most three trees, it is then straightforward to prove that deadlocks are impossible (see [15] for the details).

3.4 Leader Election Phase

In this phase, one candidate leader is finally elected to be the unique leader, provided that the shape S_0 is not unbreakably k-symmetric. If it is, then the k candidate leaders may be unable to decide who should be elected, and hence all of them become leaders.

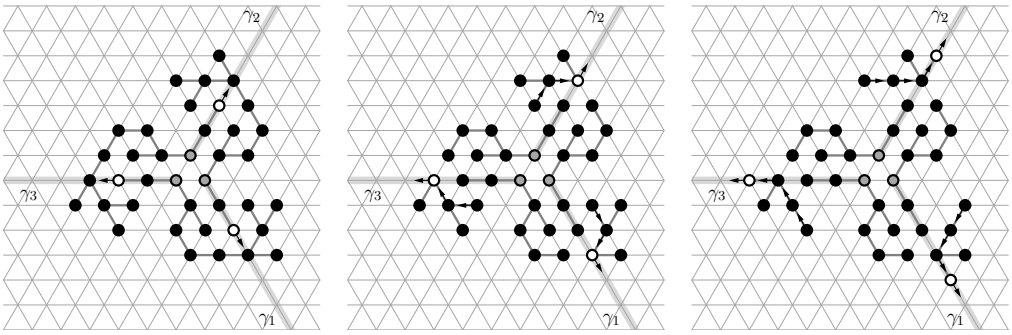
In order to elect a leader, the candidates "scan" their respective trees of the spanning forest, searching for asymmetric features of S_0 that would allow them to decide which candidate should become the leader. This task is made possible by the fact that all particles agree on the same handedness.

The algorithm is divided in steps: at each step, a particle q in each tree sends a constant-length *code* to its parent, describing its neighborhood; the message is then forwarded all the way to the candidate leader at the root of the tree. Such a code contains information on all the neighbors of q, starting from the parent and proceeding in clockwise order: the information that is encoded essentially tells whether each neighbor is a child of q, or some other particle, or an unoccupied vertex of G.

Once all candidate leaders have obtained a code from a particle in their respective tree, they send each other these codes and compare them. If the codes are not all equal, the candidates are able to elect a unique leader. Otherwise, the "election attempt" fails, and each candidate leader asks one of its children for another neighborhood code.

In the first step, the candidate leaders compare their own neighborhood codes. If the election attempt fails, each of them asks its first child in clockwise order for its neighborhood code. As the attempts keep failing, the particles in each tree are queried as in a preorder traversal, where the children of every particle are always queried in clockwise order. Since all particles agree on the same handedness, the candidate leaders keep comparing the codes of symmetric particles in their respective trees, until asymmetric particles are found. In turn, these particles have the same handedness, and so they produce the same neighborhood codes only if their surroundings are indeed symmetric.

It follows that a unique leader is not elected only if S_0 is unbreakably k-symmetric. If all the election attempts fail, the k candidate leaders have no choice but to become all leaders.



(a) Each pioneer is obstructed by a particle on its directrix.

(b) The obstructing particles detach from their parents and become the new pioneers.

(c) Each new pioneer pulls a chain of particles from the closest entry point.

Figure 4 Three stages of the straightening phase. The particles in gray are the leaders; the ones in white are the pioneers. The edges of the spanning forest are drawn in dark gray, and the arrows indicate where the particles are directed in the pulling procedure.

3.5 Straightening Phase

At the beginning of this phase, there are $k = 1$, $k = 2$, or $k = 3$ leaders, each of which is the root of a tree of particles. These k trees are rotated copies of each other, and the leaders are pairwise adjacent. The goal of this phase is to arrange the particles in a way that will make the final phase of the algorithm simpler to design: that is, k straight lines radiating from the center of the figure.

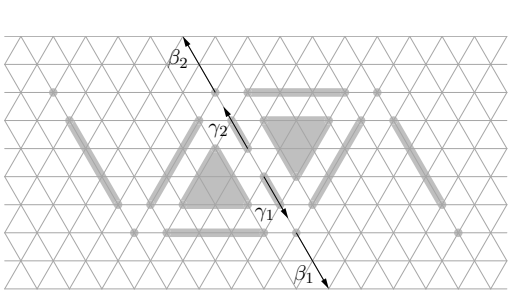
So, in this phase, each leader p_i coordinates the "straightening" of its tree. p_i chooses a ray in the plane (i.e., a half-line) as its *directrix* γ_i. By the end of the straightening phase, all particles will be located on these k directrices.

We use an important basic subroutine, called *pulling procedure*. We assume to have a *chain* Q of contracted particles, the first of which, q, is called the *pioneer*. Suppose q intends to move to an unoccupied neighboring vertex v, "pulling" the whole chain with it. So, q sends a message saying "Follow-Me" to the next particle q' in the chain Q, and then it expands to v and contracts again. When q' receives the message, it forwards it to the next particle in Q, and moves to the position previously occupied by q, and so on. When the last particle of Q has moved, it sends a message to its predecessor saying "Pulling-Complete", which is forwarded all the way to the pioneer, and the procedure ends.

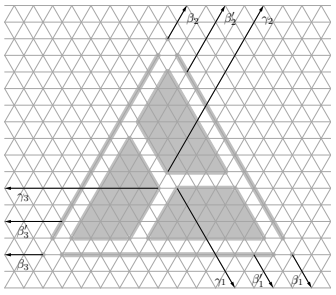
The idea of the straightening phase is that a pioneer q_i will walk along each directrix γ_i, pulling particles onto it from the tree T_i of the leader p_i (executing the pulling procedure described above). While the pioneer is doing that, the leader remains in place, except perhaps for a few stages, when it is part of a chain of particles that is being pulled by the pioneer. Eventually, all the particles of T_i will form a line segment on the directrix.

If q_i encounters another particle r on γ_i, belonging to some tree T_j, it "transfers" its role to r, and "claims" the subtree T'_j of T_j hanging from r, detaching r from its parent. The next time the new pioneer r has to pull a chain of particles, it will pull it from T'_j. For this reason, r is called an *entry point* of the directrix. This algorithm is summarized in Figure 4.

If there is only $k = 1$ leader, there are no problems with the correctness of this algorithm. Suppose that $k > 1$: then, every time a pioneer advances along its directrix, it notifies its leader, who will synchronize with the other leaders (the details are in [15]). This is to ensure that the straightening of every tree proceeds at the same pace. As a consequence, the k trees of the spanning forest, which initially are symmetric under a k-fold rotation of the plane, remain symmetric while the straightening proceeds.



(a) An unbreakably 2-symmetric shape with scale 5 with a minimal equivalent shape consisting of two adjacent faces and two edges



(b) An unbreakably 3-symmetric shape with scale 13 with a minimal equivalent shape consisting of a single face

Figure 5 Subdivision into elements (gray blobs) of unbreakably k-symmetric shapes. The directrices, the backbone, and the co-backbone are also represented.

Because of this symmetry, no conflicts between different pioneers can ever arise. For instance, it is impossible for a leaf f of a tree to be pulled along the chain led by a pioneer while another pioneer is trying to "transfer" to f. Also, the k pulling procedures that are executed in the same step involve disjoint chains: indeed, the k directrices are disjoint, and the subtrees hanging from different entry points are disjoint.

Let us prove that a system of n particles executing this phase of the algorithm performs $O(n^2)$ moves in total. Observe that each pulling procedure causes a new particle to join a directrix, and so at most n pulling procedures are performed. On the other hand, each pulling procedure involves at most n particles, and causes each of them to perform a single expansion and a single contraction. The $O(n^2)$ bound follows, and the same bound on the number of rounds can be obtained similarly.

3.6 Role Assignment Phase

At the end of the straightening phase, the system forms k equally long line segments, each of which contains a leader particle. If $k > 1$, it means that the shape S_0 that the particles originally formed was unbreakably k-symmetric. Due to Theorem 1, if this is the case, we have to assume that the "final shape" S_F that the system has to form is also unbreakably k-symmetric.

In the role assignment phase, the particles determine the scale of the shape S'_F, equivalent to S_F, that they are actually going to form. Then, each particles is assigned an identifier describing which element of S_F it is going to form in the shape composition phase.

Let S'_F be a shape equivalent to S_F. The vertices of G that lie in S'_F are partitioned into *elements* as shown in Figure 5. Each vertex of G that is in S_F corresponds to a *super-vertex* of S'_F; the interior of each edge of G contained in S_F corresponds to a *super-edge* of S'_F; the interior of each face of G contained in S_F corresponds to a *super-triangle* of S'_F. The only exceptions are the central super-edge of an unbreakably 2-symmetric shape, which is further divided into two *partial super-edges*, and the central super-triangle of an unbreakably 3-symmetric shape, which is further divided into three (trapezoidal) *partial super-triangles*.

If there are $k > 1$ leaders, they have to partition the elements of S'_F into k equal parts, so that the ith leader will form one part, $(S'_F)_i$. Assume that the similarity transformation mapping S_F to S'_F is actually a homothety centered at the center of S_0. First the ith leader defines a *backbone ray* β_i parallel to its directrix γ_i, as shown in Figure 5. If $k = 3$, it also

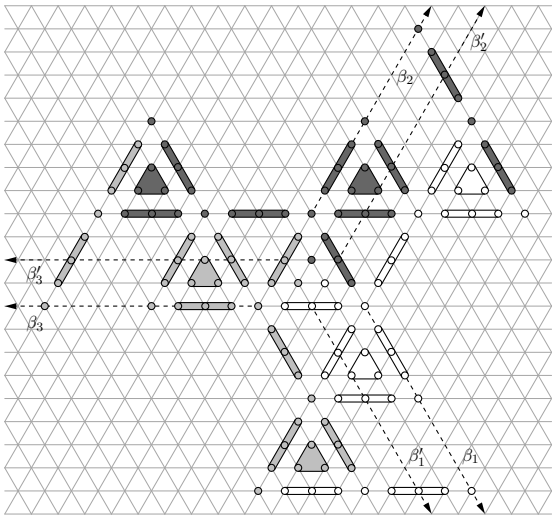


Figure 6 The elements of an unbreakably 3-symmetric shape with a possible subdivision among leaders. Blobs of the same color represent elements selected by the same leader.

defines a co-backbone β'_i, as in Figure 5b. Each leader first selects for itself all the elements of S'_F that intersect β_i or β'_i (the purpose of this selection will become clear in the next phase). Then, each leader repeatedly and deterministically selects an element of S'_F that is adjacent to an element that it has already selected and that has not been selected by any leader, yet (see [15] for further details). As a result, S'_F is partitioned into k symmetric sub-shapes $(S'_F)_i$: one possible outcome is illustrated in Figure 6. Note that, since the above algorithm is deterministic, no explicit agreement between leaders is needed.

Let us focus on a single directrix and its leader. In the role assignment phase, the leader particle repeatedly changes its internal state and transfers the leadership to a neighbor (by sending it a message). So, the particles on the directrix can collectively compute any function that is computable by a Turing machine on a tape of length n/k: the leader is the head of the machine, and the particles are the cells of its tape. Hence, with standard techniques, the leader can do computations with binary numbers using the particles as bits. In particular, it can compute a scale λ for S'_F that is large enough for it to contain all n particles, and small enough for it to be completely covered by them. Moreover, it can do so in $O(n^2)$ rounds (the details are in [15]). The number of vertices of G covered by S'_F may not be exactly n, but recall that each particle can occupy two vertices in its extended state: if n is large enough compared to the size of S_F (which is a constant m), a suitable λ can indeed be found. Some particles will have to be extended in the final configuration, and these are marked with a special *Double* flag: these will be called *double particles*.

Then, the leader assigns a constant-size *identifier* to each element of $(S'_F)_i$ (there is a constant number of them), and assigns each particle the *role identifier* corresponding to the element it will contribute to forming in the next phase. To this end, the particles are subdivided into contiguous *chunks*, each of which is associated with an element of $(S'_F)_i$ and has the appropriate size (which can be computed "indirectly" by the leader, due to the Turing-machine analogy above). If n is large enough, the leader can even place all the double particles in chunks corresponding to (partial) super-triangles (the special case in which S_F consists only of edges is discussed in [15]).

3.7 Shape Composition Phase

In this phase, each of the k leaders will guide its *team* of particles in the formation of all the elements of its sub-shape $(S'_F)_i \subseteq S'_F$. If $k > 1$, the leader preliminarily relocates its entire team from the directrix γ_i to the backbone ray β_i. This can be done with the pulling procedure used in Section 3.5 and the Turing-machine technique of Section 3.6. Indeed, the leader knows the distance between γ_i and β_i in terms of λ, but it does not have enough internal memory to count to λ. So, it can use its team as a tape and count its steps in binary.

When the team is on β_i, it starts forming the elements of $(S'_F)_i$ one by one, coordinated by the leader. First the super-edges *adjacent* to the backbone ray β_i are formed; then the super-vertices not on β_i adjacent to those super-edges, then the super-edges not on β_i adjacent to those super-vertices, etc. When all super-vertices and super-edges of $(S'_F)_i$ not on β_i have been formed, the (partial) super-triangles are formed. Finally, the elements on β_i are formed, from the closest to the center of S'_F to the farthest.

The elements on β_i are formed last because this ray is used by the leader to pull the entire "repository" of chunks wherever it has to go to form the next element e. Even though $(S'_F)_i$ may not be connected (as in Figure 6), the leader has enough particles in one chunk to count its steps while it pulls, and therefore to stop in the right position (the only exception is when only super-vertex chunks are left in the repository: this case is discussed in [15]).

Once the repository is in position to reach e, the leader moves the corresponding chunk C_e from the repository along a path W of already formed super-edges and super-vertices of $(S'_F)_i$ (by simply swapping states of particles). This is easy to do, since there are no double particles in W: they are all in the (partial) super-triangles. When C_e becomes adjacent to e, the leader pulls it into position, also pulling W (which, as a result, goes back in its place) and the rest of the repository. Then the leader leaves e and goes back to β_i through W.

Note that, when e is a (partial) super-triangle, the leader may have to use the Turing-machine technique to count its steps while forming it, so to know when it has reached a corner of e. If e contains double particles, the leader first pulls all of C_e into e, and then it leaves e. When the leader is gone, the first particle of C_e keeps pulling part of the chunk to let the double particles expand and finally cover all of e.

To prove the correctness of this algorithm, it is crucial to observe that it is impossible for two different teams to come into contact and interfere with each other's movements: this is because they are confined to move within different regions of G throughout the phase. Note that selecting all the elements that intersect its backbone ray and its co-backbone ray (see Section 3.6) gives a leader and its team free range to move between their directrix and their backbone ray during the preliminary relocation step. This is true also if one team has already started forming its elements while another is still moving from the directrix to the backbone.

Let us count the total number of moves performed in this phase. When a leader relocates its team onto the backbone, it pulls all the particles at most $O(n)$ times, and the total number of moves is at most $O(n^2)$. Then, in order to form one element of S'_F, a leader may have to pull at most $O(n)$ particles (i.e., the repository) for at most $O(n)$ times along the backbone to get the chunk into position: this yields at most $O(n^2)$ moves. Then it has to pull at most $O(n)$ particles for a number of times that is equal to the size of the element of S'_F, which is $O(n)$. Since the number of elements of S'_F is bounded by a constant, this amounts to at most $O(n^2)$ moves, again. All other operations involve only message exchanges and no movements. The $O(n^2)$ upper bound follows, and the same bound on the number of rounds is obtained similarly. (The proof of the $\Theta(m^3)$ bound on n in the theorem below is in [15].)

▶ **Theorem 3.** *Let P be a system of n particles forming a simply connected shape S_0 at stage 0. Let S_F be a shape of constant size m that is unbreakably k-symmetric if S_0 is unbreakably k-symmetric. If all particles of P execute the universal shape formation algorithm with input*

a representation of the final shape S_F, and if n is at least $\Theta(m^3)$, then there is a stage, reached after $O(n^2)$ rounds, where P forms a shape equivalent to S_F. The total number of moves performed by P up to this stage is $O(n^2)$, which is asymptotically worst-case optimal; particles no longer move afterwards.

4 Generalization to All Computable Infinite-Resolution Shapes

In the previous section we have shown that, given a shape S_F of constant size m, a system of n particles can form a shape geometrically similar to S_F (i.e., essentially a scaled-up copy of S_F) starting from any simply connected configuration S_0, provided that S_F is unbreakably k-symmetric if S_0 is, and provided that n is large enough compared to m. We only determined a bound of $\Theta(m^3)$ for the minimum n that guarantees the formability of S_F. We could improve it to $\Theta(m)$ by letting the Double particles be in any chunk and adopting a slightly more sophisticated pulling procedure in the last phase. We may wonder if this modification would make our bound optimal.

When discussing the role assignment phase, when the particles are arranged along straight lines, we have argued that the system can compute any predicate that is computable by a Turing machine on a tape of limited length. If we allow the particles to move back and forth along these lines to simulate registers, we only need a (small) constant number of particles to implement a full-fledged Turing machine with an infinite tape. So, in the role assignment phase, we are actually able to compute any Turing-computable predicate (although we would have to give up our upper bounds of $O(n^2)$ moves and rounds).

With this technique, we are not only able to replace our $\Theta(m^3)$ with the best possible asymptotic bound in terms of m, but we have a universal shape formation algorithm that, for every n and every S_F, lets the system determine if n particles are enough to form a shape geometrically similar to S_F. This is done by examining all the possible connected configurations of n particles and searching for one that matches S_F, which is of course a Turing-computable task.

Taking this idea even further, we can extend our notion of shape to its most general form. Recall that the shapes considered in [8] were sets of "full" triangles: when a shape is scaled up, all its triangles are scaled up and become larger full triangles. In this paper, we extended the notion of shape to sets of full triangles and edges: when an edge is scaled up, it remains a row of points. Of course, we can think of shapes that are not modeled by full triangles or edges, but behave like fractals when scaled up. For instance, we may want to include discretized copies of the Sierpinski triangle as "building blocks" of our shapes, alongside full triangles and edges. Scaling up these shapes causes their "resolution" to increase and makes finer details appear inside them. Clearly, the set of all the scaled-up and discretized copies of a shape made up of full triangles, edges, and Sierpinski triangles is Turing-computable.

Generalizing, we can replace our usual notion of geometric similarity between shapes with any Turing-computable equivalence relation $\sim$. Then, the shape formation problem with input a shape S_F asks to form any shape S'_F such that $S_F \sim S'_F$. This definition of shape formation problem includes and greatly generalizes the one studied in this paper, and even applies to scenarios that are not of a geometric nature. Nonetheless, this generalized problem is still solvable by particles, thanks to the technique outlined above.

References

1 H. Ando, I. Suzuki, and M. Yamashita. Formation and agreement problems for synchronous mobile robots with limited visibility. In *Proc. of ISIC*, pages 453–460, 1995.

2 S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proc. of PODC*, pages 279–288, 2016.

3 G. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of ICRA*, pages 1:449–1:455, 1994.
4 S. Das, P. Flocchini, N. Santoro, and M. Yamashita. Forming sequences of geometric patterns with oblivious mobile robots. *Distrib. Comput.*, 28(2):131–145, 2015.
5 J.J. Daymude, Z. Derakhshandeh, R. Gmyr, A. Porter, A.W. Richa, C. Scheideler, and T. Strothmann. On the runtime of universal coating for programmable matter. In *Proc. of DNA*, pages 148–164, 2016.
6 J.J. Daymude, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Improved leader election for self-organizing programmable matter. *arXiv*, 2017. URL: `https://arxiv.org/abs/1701.03616`.
7 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proc. of NanoCom*, pages 21:1–21:2, 2015.
8 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal shape formation for programmable matter. In *Proc. of SPAA*, pages 289–299, 2016.
9 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal coating for programmable matter. *Theor. Comput. Sci.*, 671:56–68, 2017.
10 Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A.W. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *Proc. of DNA*, pages 117–132, 2015.
11 S. Dolev, S. Frenkel, M. Rosenbli, P. Narayanan, and K.M. Venkateswarlu. In-vivo energy harvesting nano robots. In *Proc. of ICSEE*, pages 1–5, 2016.
12 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theor. Comput. Sci.*, 407(1):412–447, 2008.
13 N. Fujinaga, Y. Yamauchi, H. Ono, S. Kijima, and M. Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM J. Comput.*, 44(3):740–785, 2016.
14 G.A. Di Luna, P. Flocchini, G. Prencipe, N. Santoro, and G. Viglietta. Line recovery by programmable particles. In *Proc. of ICDCN*, to appear.
15 G.A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and Y. Yamauchi. Shape formation by programmable particles. *arXiv*, 2017. URL: `https://arxiv.org/abs/1705.03538`.
16 O. Michail, G. Skretas, and P.G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. In *Proc. of ICALP*, pages 136:1–136:15, 2017.
17 A. Naz, B. Piranda, J. Bourgeois, and S.C. Goldstein. A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots. In *Proc. of NCA*, pages 254–263, 2016.
18 M.J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Nat. Comput.*, 13(2):195–224, 2014.
19 P.W. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.
20 M. Rubenstein, A. Cornejo, and R. Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.
21 N. Schiefer and E. Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In *Proc. of DNA*, pages 34–54, 2015.
22 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: formation of geometric patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999.
23 T. Toffoli and N. Margolus. Programmable matter: concepts and realization. *Physica D*, 47(1):263–272, 1991.
24 J.E. Walter, J.L. Welch, and N.M. Amato. Distributed reconfiguration of metamorphic robot chains. *Distrib. Comput.*, 17(2):171–189, 2004.

Synthesis of Distributed Algorithms with Parameterized Threshold Guards*

Marijana Lazić[1], Igor Konnov[2], Josef Widder[3], and Roderick Bloem[4]

1 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`lazic@forsyte.at`
2 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`konnov@forsyte.at`
3 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`widder@forsyte.at`
4 **TU Graz, Inffeldgasse 16a/II, 8010 Graz, Austria**
`roderick.bloem@iaik.tugraz.at`

Abstract

Fault-tolerant distributed algorithms are notoriously hard to get right. In this paper we introduce an automated method that helps in that process: the designer provides specifications (the problem to be solved) and a sketch of a distributed algorithm that keeps arithmetic details unspecified. Our tool then automatically fills the missing parts.

Fault-tolerant distributed algorithms are typically parameterized, that is, they are designed to work for any number n of processes and any number t of faults, provided some resilience condition holds; e.g., $n > 3t$. In this paper we automatically synthesize distributed algorithms that work for *all* parameter values that satisfy the resilience condition. We focus on threshold-guarded distributed algorithms, where actions are taken only if a sufficiently large number of messages is received, e.g., more than t or $n/2$. Both expressions can be derived by choosing the right values for the coefficients a, b, and c, in the sketch of a threshold $a \cdot n + b \cdot t + c$. Our method takes as input a sketch of an asynchronous threshold-based fault-tolerant distributed algorithm — where the guards are missing exact coefficients — and then iteratively picks the values for the coefficients.

Our approach combines recent progress in parameterized model checking of distributed algorithms with counterexample-guided synthesis. Besides theoretical results on termination of the synthesis procedure, we experimentally evaluate our method and show that it can synthesize several distributed algorithms from the literature, e.g., Byzantine reliable broadcast and Byzantine one-step consensus. In addition, for several new variations of safety and liveness specifications, our tool generates new distributed algorithms.

1998 ACM Subject Classification F.3.1 [*Logic and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs, D.4.5 [*Software*]: Operating systems: Fault-tolerance, Verification

Keywords and phrases fault-tolerant distributed algorithms, byzantine faults, parameterized model checking, program synthesis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.32

* Supported by: the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403, S11405, and S11406), project PRAVDA (P27722), and Doctoral College LogiCS (W1255-N23); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103). The computational results presented have been achieved [in part] using the Vienna Scientific Cluster (VSC).

1 Introduction

Design and implementation of parameterized fault-tolerant distributed systems are error-prone tasks. There is a mature *theory* regarding mathematical proof methods, which found their way into formal frameworks like I/O Automata [24] and TLA+ [22]. Recent approaches [17, 23, 31] provide tool support to establish correctness of implementations, by manually constructing proofs with an interactive theorem prover. Although, if successful, this approach provides a machine-checkable proof [9, 4], it requires huge manual efforts from the user. A logic for distributed consensus algorithms in the HO Model [8] was introduced in [13], which allows one to automatically check the invariants (for safety) and ranking functions (for liveness), that is, the manual effort is reduced to finding right invariants and ranking functions. Model checking of distributed algorithms promises a higher degree of automation. For consensus algorithms in the HO Model, the results of [25] reduce the verification to checking small systems of five or seven processes. For the asynchronous model, an efficient model checking technique for threshold-guarded distributed algorithms was introduced in [20, 19]. Notably, this technique verifies both safety and liveness properties. In all these methods, the user has to produce an implementation (or design), and the goal is to check (using techniques that vary in the degree of automation) whether this implementation satisfies a given specification.

In this paper we explore synthesis as it promises even more automation. The user just provides required properties and a sketch of an asynchronous algorithm, and our tool automatically finds a correct distributed algorithm. In this way we generate new fault-tolerant algorithms that are *correct by construction*. In our experiments we first focus on existing specifications [29, 7, 30, 28] from the literature, in order to be able to compare the output of our tool with known algorithms. We then give new variations of safety and liveness specifications, and our tool generates new distributed algorithms for them.

Parametrized synthesis. Similar to the verification approaches above, we are interested in the parameterized version of the problem: Rather than synthesizing a distributed algorithm that consists of, say, four processes and tolerates one fault, our goal is to synthesize an algorithm that works for n processes, out of which t may fail, for all values of n and t that satisfy a resilience condition, e.g., $n > 3t$. This is in contrast to recent work on synthesis of fault-tolerant distributed algorithms [16, 15, 14] that requires the user to fix the number of processes; typically to some $n < 10$. In some special cases, manual arguments or cut-off theorems generalize synthesis results for small systems to parameterized ones [5, 12, 26]. However, similar to parameterized verification [2, 6], the parameterized synthesis problem is in general undecidable [18]. As in the parameterized verification approach of [19], we will therefore limit ourselves to a specific class of distributed algorithms, namely, *threshold-guarded distributed algorithms*. These thresholds are arithmetic expressions over parameters, e.g., $n/2$, and determine for how many messages processes should wait (a majority in the example).

More specifically, the user provides as input a distributed algorithm with holes as in Figure 1: The user defines the control flow, and keeps the threshold expressions — noted as τ_{0toSE} and τ_{AC} in the figure — unspecified. As pseudo code has no formal semantics, it cannot be used as a tool input. Rather, our tool takes as input a sketch threshold automaton.

▶ **Example 1.** Figure 1 is a pseudo code representation of the input, and Figure 2 shows the corresponding sketch threshold automaton; they are related as follows. The initial locations ℓ_0 and ℓ_1 of the sketch threshold automaton in Figure 2 correspond to initial states in Figure 1 where *myval* is equal to 0 and 1, respectively. Edges are labeled by $g \mapsto \texttt{act}$, where expression g is a threshold guard, and the action `act` may increment a shared variable.

Code of a correct process i:
var $myval_i \in \{0,1\}$
var $accept_i \in \{false, true\} \leftarrow false$

while *true* **do** *(in one step)*
if $myval_i = 1$
and not sent ECHO before
then send ECHO **to** all

if received ECHO **from** $\geq \tau_{\mathsf{0toSE}}$ **distinct** processes
and not sent ECHO before
then send ECHO **to** all

if received ECHO **from** $\geq \tau_{\mathsf{AC}}$ **distinct** processes
then $accept_i \leftarrow true$
od

Figure 1 A single-round version of the reliable broadcast algorithm [29] with holes.

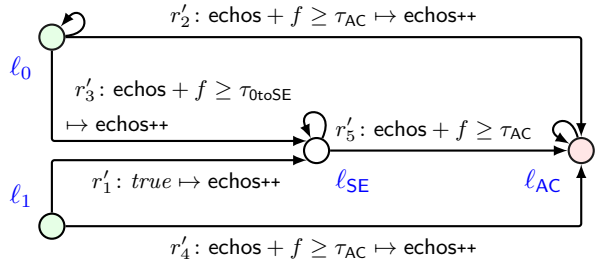


Figure 2 A sketch threshold automaton.

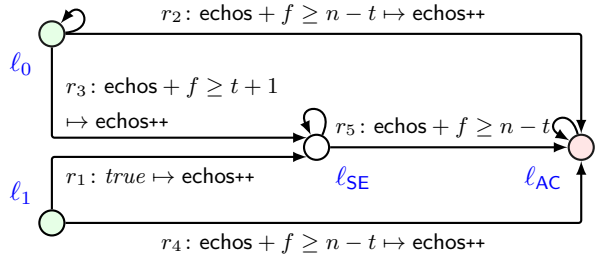


Figure 3 A synthesized threshold automaton.

The action echos++ corresponds to the pseudo-code statement: `send <echo> to all`. (The message buffers are replaced by a shared variable that is increased whenever a message is sent. This typically can be done for algorithms that only count messages, and do not distinguish the senders. For instance, a bisimulation between models with message buffers and shared variables was proven in [21].) By going to local state ℓ_{SE}, a process records that it has sent *echo*. Finally, by going to the local state ℓ_{AC}, a process records that it has set accept to true. The "$+f$" terms in threshold guards model that messages from up to f Byzantine processes may be received ($f \leq t$), while we model only the $n - f$ correct processes explicitly.

We use self loops to capture the behavior that processes may take arbitrarily many steps before receiving a message sent to them, that is, asynchronous communication. For instance, the self loop in ℓ_0 allows processes to stay in ℓ_0 even if the guards of the other outgoing edges evaluate to true. That every message from a correct process is eventually received is a fairness constraint and is therefore not part of the threshold automaton, but is captured in the specifications. For instance, such a fairness constraint would be that if echos $\geq n - t$, then every process must eventually leave location ℓ_0. (In the fairness constraint, the "$+f$" does not appear, because messages from faulty processes are not guaranteed to arrive.)

The "holes" τ_{0toSE} and τ_{AC} in Figure 2 are the missing thresholds, which should be linear combination of the parameters n and t. Therefore τ_{0toSE} has the form $\mathbf{?}_1 \cdot n + \mathbf{?}_2 \cdot t + \mathbf{?}_3$, and τ_{AC} has the form $\mathbf{?}_4 \cdot n + \mathbf{?}_5 \cdot t + \mathbf{?}_6$. The unknown coefficients $\mathbf{?}_i$, for $1 \leq i \leq 6$, have to be found by the synthesis tool. One solution is $\mathbf{?}_1 = 0$, $\mathbf{?}_2 = 1$, $\mathbf{?}_3 = 1$, $\mathbf{?}_4 = 1$, $\mathbf{?}_5 = -1$, and $\mathbf{?}_6 = 0$, that is, $\tau_{\mathsf{0toSE}} = t + 1$ and $\tau_{\mathsf{AC}} = n - t$. This solution is depicted in Figure 3. ◁

In addition to a sketch threshold automaton, the user has to provide a specification, that is, safety and liveness properties the distributed algorithm should satisfy. Based on these inputs, our tool generates the required coefficients, that is, a threshold automaton as in Figure 3. The synthesis approach of this paper is enabled by a recent advance [19] in parameterized model checking of *safety and liveness* properties of distributed algorithms.

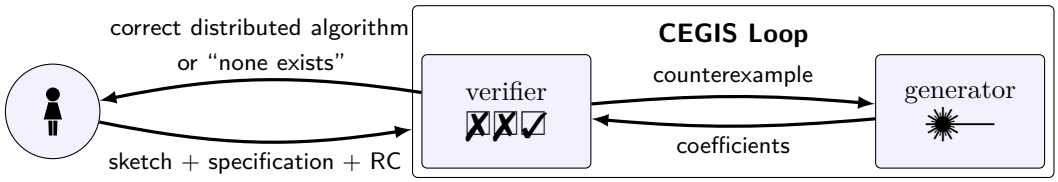


Figure 4 The synthesis loop implemented in this paper.

Existing model checking engine. The central idea of the verification approach in [19] is to formalize a distributed algorithm as counter system defined by a threshold automaton. A state of a counter system records how many processes are in which local state (e.g., $\ell_0, \ell_1, \ell_{\mathsf{SE}}, \ell_{\mathsf{AC}}$). A transition checks the guard and decreases or increases the related process counters. A transition can be written as a set of constraints in linear integer arithmetic LIA. (Threshold guards with rational coefficients, e.g., $\mathsf{echos} > \frac{n}{2}$, can be converted to integer constraints, e.g., $2 \cdot \mathsf{echos} > n$.) Thus, one can check for existence of specific executions by using SMT solvers, which extend SAT solvers (for Boolean satisfiability) with first-order theories, in our case, LIA. As shown in [20, 19], resilience conditions, executions of threshold-guarded distributed algorithms, and specifications can be encoded as logical formulas, whose satisfiability can be checked by solvers such as Z3 [11] and CVC4 [3]. In particular, the queries used in [20, 19] correspond to counterexamples to a specification: If the SMT solver finds all queries to be unsatisfiable, the distributed algorithm is correct. Otherwise, if a query is satisfiable, the SMT solver outputs a satisfying assignment, that is an error trace, called *counterexample*.

The synthesis approach of this paper. Figure 4 gives an overview of our method that takes as input (i) a sketch of a distributed algorithm, (ii) a set of safety and liveness specifications, and (iii) a resilience condition like $n > 3t$, and produces as output a correct distributed algorithm, or informs the user that none exist.

We follow the CEGIS approach to synthesis [1], which proceeds in a refinement loop. Roughly speaking, the verifier starts by picking default values for the missing coefficients — e.g., a vector of zeroes — and checks whether the algorithm is correct with these coefficients. Typically this is not the case and the *verifier* produces a counterexample. By automatically analyzing this counterexample, the *generator* learns constraints on the coefficients that are known to produce counterexamples. The generator gives these constraints to an SMT solver that generates new values for the coefficients, which are used in a new verifier run. If the verifier eventually reports that the current coefficients induce a correct distributed algorithm, we output this algorithm. The theory from [19] then implies correctness of the algorithm.

Termination of synthesis. The remaining theoretical problem that we address in this paper is termination of the refinement loop: In principle, the generator can produce infinitely many vectors of coefficients. In case there is no solution (which is typically the case in Byzantine fault tolerance if $n \leq 3t$), the naïve approach from the previous paragraph does not terminate, unless we restrict the guards to "reasonable" values. In this paper, we require the guards to lie in the interval $[0, n]$. We call such guards *sane*. For instance, although syntactically the expressions $\mathsf{echos} \leq -42n$ and $\mathsf{echos} > 2n$ are threshold guards, they are not sane, while $\mathsf{echos} \geq t + 1$ is sane. We mathematically prove that all sane guards of a specific structure have coefficients within a hyperrectangle. We call this hypperrectangle a *sanity box*, and prove that its boundaries depend only on the resilience condition. Within the sanity box, there is only a finite number of coefficients, if we restrict them to integers or rationals with a

fixed denominator. We thus obtain a finite search space and a completeness result for the synthesis loop.

Safety, liveness, and the fraction of faults. We consider the conjunction of *safety* and *liveness* specifications, as these specifications in isolation typically have trivial solutions; e.g., "do nothing" is always safe. If just given a safety specification, our tool generates thresholds like n for all guards, which leads to all guards evaluating to false initially. Hence, no action can ever be taken, which is a valid solution if liveness is not required.

Besides, our tool treats resilience conditions precisely. On the one hand, given the sketch from Figure 2, and the resilience condition $n > 3t$, in a few seconds our tool generates the threshold automaton in Figure 3. On the other hand, in the case of $n \geq 3t$, our tool reports (also within seconds) that no such algorithm exists, which in fact constitutes an automatically generated impossibility result for sane thresholds and a fixed sketch.

Experimental evaluation. We extended the tool ByMC [20] with our technique and conducted experiments based on the freely available benchmarks from [20]: folklore reliable broadcast [7], consistent broadcast [29, 30], and one-step Byzantine asynchronous consensus BOSCO [28]. For these benchmarks, we replaced the threshold guards by threshold guards with holes. By experimental evaluation, we show that our method can be used to generate coefficients even for quite intricate fault-tolerant distributed algorithms that tolerate Byzantine faults. In particular BOSCO proved to be a hard instance. It has to satisfy constraints derived from different safety an liveness specifications under different resilience conditions $n > 3t$, $n > 5t$, and $n > 7t$. Our tool is able to derive the three different threshold guards the algorithm requires. Finally, we give variations of specifications, and synthesize distributed algorithms from them that have not been produced before.

2 Modelling Threshold-Guarded Distributed Algorithms

Threshold-guarded algorithms are formalized by threshold automata. We recall the notions of threshold automata [19] and introduce the new concept of sketches. As usual, $\mathbb{N}_0$ is the set of natural numbers including 0, and $\mathbb{Q}$ is the set of rational numbers. The set Π is a finite set of *parameter variables* that range over $\mathbb{N}_0$. Typically, Π consists of three variables: n for the total number of processes, f for the number of actual faults in a run, and t for an upper bound on f. The parameter variables from Π are usually restricted to admissible combinations by a formula that is called a *resilience condition*, e.g., $n > 3t \wedge t \geq f \geq 0$. The set Γ is a finite set that contains *shared variables* that store the number of distinct messages sent by distinct (correct) processes, the variables in Γ also range over $\mathbb{N}_0$. In the example in Figure 3, $\Gamma = \{\mathsf{echos}\}$. For the variables from Γ, we will use names echos, x, y, etc.

For a set of variables V, a function $\nu : V \to \mathbb{Q}$ is called *an assignment*; its domain V is denoted with $dom(\nu)$. In this paper, we use Φ, Ψ, and Θ for first-order logic (FOL) formulas; e.g., when encoding linear integer constraints in SMT. For a FOL formula Φ, we write $free(\Phi)$ for the set of Φ's free variables, that is, the variables not bound with a quantifier. (For convenience, we assume that quantified variables have unique names and they are different from the names of the free variables.) Given an assignment $\nu : V \to \mathbb{Q}$ and a FOL formula Φ, we define a *substitution* $\Phi[\nu]$ as a FOL formula that is obtained from Φ by replacing all the variables from $V \cap free(\Phi)$ with their values in ν.

To introduce sketches of threshold automata — such as in Figure 2 — we define unknowns such as $\mathbf{?}_1$. The set U is a finite set of *unknowns* that range over $\mathbb{Q}$. For the variables from U, we use the names $\mathbf{?}_1$, $\mathbf{?}_2$, etc. We denote the rational values of unknowns with a, b, c, etc.

Generalized threshold guards, or just *guards*, are defined according to the grammar:

$$\begin{aligned}
&\mathit{Guard} ::= \mathit{Shared} \geq \mathit{LinForm} \mid \mathit{Shared} < \mathit{LinForm} && \mathit{Shared} ::= \langle\text{variable from } \Gamma\rangle\\
&\mathit{LinForm} ::= \mathit{FreeCoeff} \mid \mathit{Prod} \mid \mathit{Prod} + \mathit{LinForm} && \mathit{Param} ::= \langle\text{a variable from } \Pi\rangle\\
&\mathit{FreeCoeff} ::= \mathit{Rat} \mid \mathit{Unknown} && \mathit{Unknown} ::= \langle\text{a variable from } U\rangle\\
&\mathit{Prod} ::= \mathit{Rat} \times \mathit{Param} \mid \mathit{Unknown} \times \mathit{Param} && \mathit{Rat} ::= \langle\text{a rational from } \mathbb{Q}\rangle
\end{aligned}$$

For convenience, we assume that every parameter appears in *LinForm* at most once. Let $\bar{\pi}$ denote the vector $(\pi_1, \ldots, \pi_{|\Pi|}, 1)$ that contains all the parameter variables from Π in a fixed order as well as number 1 as the last element. Then, every generalized guard can be written in one of the two following forms $x \geq \bar{u} \cdot \bar{\pi}^\intercal$ or $x < \bar{u} \cdot \bar{\pi}^\intercal$, where x is a shared variable from Γ, and $\bar{u}$ is a vector of elements from $U \cup \mathbb{Q}$. When a parameter does not appear in a generalized guard, its corresponding component in $\bar{u}$ equals zero. We say that a guard is a *sketch guard* if its vector $\bar{u}$ contains a variable from U. A guard that is not a sketch guard is called a *fixed guard*. Previous work [19] was only concerned with fixed guards.

Since threshold guards are a special case of FOL formulas, we can apply substitutions to them. For instance, given an assignment $\nu : U \to \mathbb{Q}$ and a threshold guard g, the substitution $g[\nu]$ replaces every occurence of an unknown $\mathbf{?}_i \in U$ in g with the rational $\nu(\mathbf{?}_i)$.

Threshold automata, denoted by TA, are edge-labeled graphs, where vertices are called *locations*, and edges are called *rules*. Rules are labeled by $g \mapsto \texttt{act}$, where expression g is a fixed threshold guard, and the action `act` may increment a shared variable. We define *generalized threshold automata* GTA, in the same way as threshold automata, with the only difference that expressions g in the edge labeling are generalized threshold guards. If all generalized guards in a GTA are fixed, then that GTA is a TA. If at least one of the edges of a GTA is labeled by a sketch guard, then we call this automaton a *sketch threshold automaton*, and we denote it by STA. Given an STA and an assignment $\nu : U \to \mathbb{Q}$, we obtain a threshold automaton STA$[\nu]$ by applying substitution $g[\nu]$ to every sketch guard g in STA.

Counter systems. Executions of threshold automata are formalized as counter systems. Since processes just wait for messages until a threshold is reached and do not distinguish the senders, the systems we consider are symmetric. This allows us to represent a global state — a configuration — by (process) counters: Instead of recording which process is in which local state (which is done typically in distributed algorithms theory), we capture for each local state, how many processes are in it, and then use the rules of the threshold automaton to define the transitions between configurations. In the following, we quickly sketch the semantics to the extent necessary for this paper. Complete definitions can be found in [19].

For every TA we define a counter system as a transition system. First, for every location ℓ we introduce a counter $\boldsymbol{\kappa}[\ell]$ that keeps track of the number of processes in that particular location. A configuration σ is defined as an assignment of all counters of locations, all shared variables from Γ, and all parameters from Π, that respects the resilience condition. If a rule r is an edge (ℓ, ℓ') of a TA, then a transition (r, m) represents m processes moving from the location ℓ to ℓ'. We call m the *acceleration factor*. If $m = 1$ for all transitions in an execution, we get asynchronous executions where one process moves at a time, that is, interleaving semantics. If the rule r has a label $g \mapsto \texttt{act}$, then (r, m) can be applied only in a configuration in which the counter $\boldsymbol{\kappa}[\ell]$ has a value at least m, and g evaluates to true, and remains true during $m - 1$ applications of `act`. In other words, only if the threshold from g is reached, and there are enough processes in location ℓ; see [19] for details. After executing the transition (r, m), counters are updated such that $\boldsymbol{\kappa}[\ell]$ is decreased by m and $\boldsymbol{\kappa}[\ell']$ is increased by m, and shared variables are updated according to the action `act`, m times.

▶ **Example 2.** Consider the TA from Figure 3. One configuration is the following: parameters are $n = 7$, $f = t = 2$, satisfying resilience condition $n > 3t \geq 0 \wedge f \leq t$, counters have values $\boldsymbol{\kappa}[\ell_0] = 2$, $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 3$, $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_{\mathsf{AC}}] = 0$ and shared variable $\mathsf{echos} = 3$. (As we only model correct process explicitly, the counters add up to $n - f = 5$.) As in this configuration we have that $\mathsf{echos} + f = 5 \geq 5 = n - t$, and $\boldsymbol{\kappa}[\ell_0] = 2$, we can execute transition $(r_2, 2)$. The obtained configuration has the same parameter values, but counters are changed: $\boldsymbol{\kappa}[\ell_0] = \boldsymbol{\kappa}[\ell_1] = 0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 3$, and $\boldsymbol{\kappa}[\ell_{\mathsf{AC}}] = 2$. Also, as the action of the rule r_2 is $\mathsf{echos}{+}{+}$, and two processes are moving along this edge, then the new value of echos is 5. ◁

With a TA we associate a set of predicates $\mathcal{P}_{\mathsf{TA}}$ that track properties of the system states. The set $\mathcal{P}_{\mathsf{TA}}$ consists of the TA's threshold guards and a test $\boldsymbol{\kappa}[\ell] = 0$ for every location ℓ in TA. For every configuration σ, one can compute the set $\rho(\sigma) \subseteq \mathcal{P}_{\mathsf{TA}}$ of the predicates that hold true in σ. As was demonstrated in [19], the predicates from $\mathcal{P}_{\mathsf{TA}}$ and linear temporal logic are sufficient to express the safety and liveness properties of threshold-guarded distributed algorithms found in the literature. Essentially, the test $\boldsymbol{\kappa}[\ell] = 0$ evaluates to true if no process is in location ℓ, and $\boldsymbol{\kappa}[\ell] \neq 0$ evaluates to true if there is at least one process at ℓ. That all processes are in specific locations can be expressed by a condition that states that "no processes are in the other locations", that is, as a Boolean combination of tests for zero. We include the threshold guards in $\mathcal{P}_{\mathsf{TA}}$ to be able to express the fairness properties such as: if $\mathsf{echos} \geq t + 1$, then every process should eventually make one of the transitions labelled with $\mathsf{echos} + f \geq t + 1$. Examples of such properties for our benchmarks are given in Section 5.

A system execution is expressed as a path in the counter system. Formally, a *path* is an infinite alternating sequence of configurations and transitions, that is, $\sigma_0, t_1, \sigma_1, \ldots, t_i, \sigma_i, \ldots$, where σ_0 is an initial configuration, and σ_{i+1} is the result of applying t_{i+1} to σ_i for $i \geq 0$. The infinite sequence $\rho(\sigma_0), \rho(\sigma_1), \ldots$ is called the path *trace*. With $\mathsf{Traces}_{\mathsf{TA}}$ we denote the set of all path traces in the TA's counter system. Correctness of a distributed algorithm then means that all traces in $\mathsf{Traces}_{\mathsf{TA}}$ satisfy a specification expressed in linear temporal logic [10]. The verification approach from [19] discussed in Section 3 specifically looks for traces that violate the specification. Such traces are characterized by the temporal logic $\mathsf{ELTL}_{\mathsf{FT}}$ that allows one to express *negations of specifications* relevant for fault-tolerant distributed algorithms.

3 Verification machinery

In [19] we introduced a technique for parameterized verification of threshold-based distributed algorithms. Given a fixed threshold automaton TA, a resilience condition RC, and a set $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas representing negation of specifications, we check whether there is an execution violating the specification $(\varphi_1 \wedge \ldots \wedge \varphi_k)$. Thus, as an output, the algorithm from [19] either confirms correctness, or gives a counterexample. In this paper, we use this technique as a black box, that is, we assume that there is a function

$$\mathsf{verify}_{\mathsf{ByMC}}(\mathsf{TA}, RC, \{\neg\varphi_1, \ldots, \neg\varphi_k\})$$

that either reports a counterexample, or that TA is correct. As our synthesis approach learns from counterexamples, we recall the form of the counterexamples reported by the verifier.

Representative executions and schemas. The idea of [19] lies in automatically computing length and structure of representative executions in advance, whose shape we call schemas. A *schema* is an alternating sequence of contexts (sets of guards) and sequences of rules. Precise definition of a schema and its encoding can be found in [20, 19]. Intuitively, a schema is a concatenation of multiple *simple* schemas. A simple schema has the form

Algorithm 1 Pseudo-code of the synthesis loop.

```
1  procedure synt_ByMC(STA, RC, {¬φ_1, ..., ¬φ_k})
2    Θ_0 := bound_U(RC) and i := 0
3    while (true)
4      call check_SMT(Θ_i)
5      case unsat ⇒ print 'no more solutions' and exit()
6      case sat(μ) ⇒ /* μ assigns rationals to the variables in U */
7        call verify_ByMC(STA[μ], RC, {¬φ_1, ..., ¬φ_k})
8        case correct ⇒
9          print 'solution μ' /* exclude this solution and continue */
10         Θ_{i+1} := Θ_i ∧ ⋁_{?_j ∈ U} ?_j ≠ μ[?_j] and i := i+1
11       case counterexample(S, ν) ⇒ /* dom(ν) ∩ U = ∅ */
12         S_U := generalize(S, STA)
13         Ψ := formula_SMT(S_U)
14         Θ_{i+1} := Θ_i ∧ ¬Ψ[ν] and i := i+1
```

$\{g_1, \ldots, g_k\}\, r_1 \ldots r_s\, \{g'_1, \ldots, g'_{k'}\}$, where $k, k', s \in \mathbb{N}$, $g_1, \ldots, g_k, g'_1, \ldots, g'_{k'}$ are guards, and $r_1, \ldots, r_s$ are rules. Given acceleration factors m_i, $1 \leq i \leq s$, such that $0 \leq m_i \leq n$, the simple schema generates an execution where all the guards $g_1, \ldots, g_k$ hold in its initial configuration, and after executing $(r_1, m_1), \ldots, (r_s, m_s)$, we arrive in a configuration where all the guards $g'_1, \ldots, g'_{k'}$ hold. As proven in [19], specific schemas of fixed length represent infinite executions (that end in an infinite loop) as required for counterexamples to liveness.

Our verification tool considers each schema in isolation, and basically searches for an evaluation ν of the parameters $(n,\ t,\ f)$, an initial configuration (values of counters of initial local states), and all the acceleration factors, such that the resulting execution is admissible (only enabled rules are executed, etc.). Such an execution — if found — constitutes a counterexample, and the tool reports the corresponding pair (schema, ν).

Solver. Our tool encodes a schema as an SMT formula over parameters, counters of local states, global variables, and acceleration factors. This formula is a conjunction of equalities and inequalities in linear integer arithmetic. Inequalities come from guards, and have shared variables and parameters as free variables. Equalities come from transitions, as every transition is encoded as updating counters of local states and shared variables. The tool ByMC [19] calls an SMT solver to check satisfiability of the formula.

4 Synthesis

Synthesis problem. A temporal logic formula φ in $\mathsf{ELTL_{FT}}$ describes an (infinite) set of bad traces that the synthesized algorithm must avoid. Therefore, we consider the following formulation of the *synthesis problem*. Given a sketch threshold automaton STA and an (infinite) set of bad traces $\mathsf{Traces_{Bad}}$, either:

- find an assignment $\mu : U \to \mathbb{Q}$, in order to obtain the fixed threshold automaton $\mathsf{STA}[\mu]$ whose traces $\mathsf{Traces_{STA}}[\mu]$ *do not intersect* with $\mathsf{Traces_{Bad}}$, or
- report that no such assignment exists.

Our approach is to find values for the unknowns in a synthesis refinement loop and test them with the verification technique from Section 3.

Synthesis loop. Algorithm 1 shows the pseudo-code of the synthesis procedure $\mathsf{synt}_{\mathsf{ByMC}}$. At its input the procedure receives a sketch threshold automaton STA, a resilience condition, and a set of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$, which capture the bad traces $\mathsf{Traces}_{\mathsf{Bad}}$. In line 2, formula Θ_0, which captures constraints on the unknowns from U, is initialized using a function $\mathtt{bound}_U$. In principle, $\mathtt{bound}_U$ can be initialized to true (no constraints). However, to ensure termination, we will discuss later in this section, how we obtain constraints that bound the coefficients of sane guards. After initialization we enter the synthesis loop.

The SMT solver checks whether Θ_i has a satisfying assignment to the unknowns in U (line 4). If Θ_i is unsatisfiable, the loop terminates with a *negative outcome* in line 5. Otherwise, the SMT solver gives us an assignment $\mu : U \to \mathbb{Q}$ that is a solution candidate. To check feasibility of μ, the verifier is called for the fixed threshold automaton $\mathsf{STA}[\mu]$ in line 7. The verifier generates multiple schemas, each being one SMT query, which are checked either sequentially or in parallel. If the verifier reports that a schema that produces a counterexample does not exist, then the candidate assignment μ and threshold automaton $\mathsf{STA}[\mu]$ give us a solution to the synthesis problem. If we were interested in just one solution, the loop would terminate here with a *positive outcome*. However, because we want to enumerate all solutions, our function does a complete search, such that we exclude the solution μ for the future search in line 10, and continue.

If the verifier finds a counterexample, the loop proceeds with the branch in line 11. A counterexample is a schema S of $\mathsf{STA}[\mu]$ and a satisfying assignment $\nu : V \to \mathbb{Q}$ to the free variables V of the SMT formula $\mathsf{formula}_{\mathsf{SMT}}$, which include the parameters Π, shared variables x^j for $x \in \Gamma$, and counters $\boldsymbol{\kappa}^j[\ell]$ for each local state $\ell \in \mathcal{L}$ and every configuration j. In principle, we could exclude μ from consideration similar to line 10. For efficiency, we want to exclude a larger set of evaluations, namely all that lead to the same counterexample: We produce a *generalized* schema S_U, by replacing the rules and guards in S, which belong to the threshold automaton $\mathsf{STA}[\mu]$ with the rules and guards of the sketch threshold automaton STA (line 12). In line 13, we generate a generalized counterexample Ψ. As Ψ is derived from a counterexample with valuations μ and ν, we know that $\Psi[\nu][\mu]$ is true. Further, for every evaluation of the unknowns μ', if $\Psi[\nu][\mu']$ is true, then $\Psi[\nu][\mu']$ is a counterexample. To exclude all these evaluations μ' at once, we conjoin $\neg\Psi[\nu]$ with Θ_i in line 14, which gives us new constraints on the unknowns, before entering the next loop iteration.

The synthesis loop terminates only in line 5, that is, if Θ_i is unsatisfiable. As, in this case, Θ_i is equivalent to false, the following observation guarantees that all satisfying assignments of Θ_0 have been explored and all solutions (if any exists) have been reported.

▶ **Observation 3.** *At the beginning of every iteration $i \geq 0$ of the synthesis loop in lines 3–14, the following invariant holds: if $\mu : U \to \mathbb{Q}$ is a satisfying assignment of formula $\Theta_0 \wedge \neg\Theta_i$, then either: (1) μ was previously reported as a solution in line 9, or (2) μ was previously excluded in line 14 and thus is not a solution.* ◁

Completeness and termination for sane guards. Without restricting Θ_0, the search space for coefficients is infinite. In the following, we show that restricting the synthesis problem to sane guards bounds the search space.

The role of threshold guards is typically to check whether the number of distinct senders, from which messages are received, reaches a threshold. We also use threshold guards in our models to bound the number of processes that go into a special crash state. In both cases, one counts distinct processes and it is therefore natural to consider only those thresholds

whose value is in $[0, n]$. More precisely, if the guard has a form $x \geq \bar{u} \cdot \bar{\pi}^\intercal$ or $x < \bar{u} \cdot \bar{\pi}^\intercal$, then for all parameter values that satisfy resilience condition it holds that $0 \leq \bar{u} \cdot \bar{\pi}^\intercal \leq n$. We call such guards *sane* for a given resilience condition.

Theorem 4 considers a general case of hybrid failure models [30] where different failure bounds exist for different failure models (e.g., t_1 Byzantine faults and t_2 crash faults), and these failure bounds are related to the number of processes n by a resilience condition[1] of the form $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$. We bound the values of the coefficients of sane guards.

▶ **Theorem 4.** *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, for $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{or} \quad x < an + (b_1 t_1 + \ldots + b_k t_k) + c,$$

where $x \in \Gamma$, and $a, b_1, \ldots, b_k, c \in \mathbb{Q}$. If the guard is sane for the resilience condition, then

$$0 \leq\ a\ \leq 1, \tag{1}$$

$$-\delta_i - 1 <\ b_i\ < \delta_i + 1,\ \textit{for all } i = 1, \ldots, k \tag{2}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq\ c\ \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{3}$$

The case when $k = 1$ gives us the classical resilience condition where the system model assumes *one* type of faults (e.g., crash), and the assumed number of faults t is related to the total number of processes n, by a condition $n > \delta t \geq 0$ for some $\delta > 0$. If the guard that compares a shared variable and $an + bt + c$ is sane for the resilience condition, then we obtain that $0 \leq a \leq 1$, $-\delta - 1 < b < \delta + 1$, and $-2\delta - 2 \leq c \leq 2\delta + 2$. Any restriction of the intervals from Theorem 4 to finite sets gives us completeness: If we reduce the domain of variables from U to integers, or to rationals with fixed denominator (e.g., $\frac{z}{10}$ for $z \in \mathbb{Z}$), one reduces the search space to a finite set of valuations. All threshold-based distributed algorithms we are aware of, use guards with coefficients that are either integers or rationals with a denominator not greater than 3. Thus, we restrict our intervals by intersecting them with the set of rational numbers whose denominator is at most D, for a given $D \in \mathbb{N}$.

The following corollary is a direct consequence of Theorem 4, and it tells us how to modify intervals if the coefficients are rational numbers with a fixed denominator.

▶ **Corollary 5.** *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k\right) + \frac{\tilde{c}}{D} \quad \text{or} \quad x < \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k\right) + \frac{\tilde{c}}{D},$$

where $x \in \Gamma$, $\tilde{a}, \tilde{b_1}, \ldots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$, $D \in \mathbb{N}$. If the guard is sane for the resilience condition then

$$0 \leq\ \tilde{a}\ \leq D, \tag{4}$$

$$D(-\delta_i - 1) <\ \tilde{b_i}\ < D(\delta_i + 1),\ \textit{for all } i = 1, \ldots, k, \tag{5}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k - 1) \leq\ \tilde{c}\ \leq D(2(\delta_1 + \ldots + \delta_k) + k + 1). \tag{6}$$

[1] Because a guard that is sane for a weaker resilience condition, is also sane for a stronger one, Theorem 4 and Corollary 5 also hold for any resilience condition that follows from this one, e.g., $n > \max\{\delta_1 t_1, \ldots, \delta_k t_k\} \wedge \forall i.\ t_i \geq 0$. We can use the same intervals, confirmed by the same proofs as in Appendix A. However, our benchmarks use the form of resilience conditions of Theorem 4.

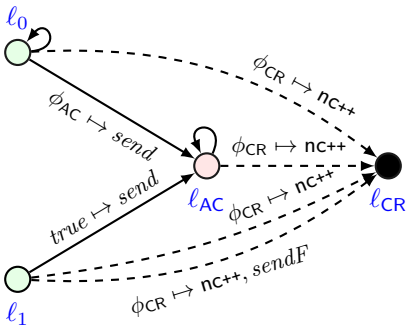


Figure 5 A sketch threshold automaton for folklore reliable broadcast.

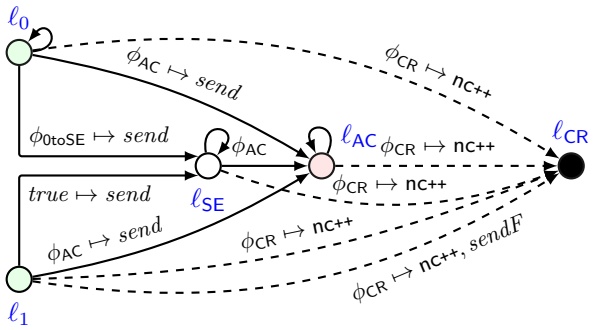


Figure 6 A sketch threshold automaton for reliable broadcast with Byzantine and crash faults.

Constraints (4)–(6) constitute the sanity box that function bound_U computes in Algorithm 1. By fixing D, we restrict Θ_0 to have finitely many satisfying assignments (integers). Hence, the loop terminates. Statements similar to Theorem 4 and Corollary 5 can be derived for other forms of threshold guards, e.g., for thresholds with floor or ceiling functions.[2]

5 Case Studies and Experiments

We have extended ByMC [20, 19] with the synthesis technique presented in this paper. A virtual machine with the tool and the benchmarks is available from: `http://forsyte.at/software/bymc`.[3] ByMC is written in OCaml and uses Z3 [11] as a backend SMT solver. We ran the experiments on two systems: a laptop and the Vienna Scientific Cluster (VSC-3). The laptop is equipped with 16 GB of RAM and Intel® Core™ i5-6300U processor with 4 cores, 2.4 GHz. The cluster VSC-3 consists of 2020 nodes, each equipped with 64 GB of RAM and 2 processors (Intel® Xeon™ E5-2650v2, 2.6 GHz, 8 cores) and is internally connected with an Intel QDR-80 dual-link high-speed InfiniBand fabric: `http://vsc.ac.at`.

We synthesize thresholds for asynchronous fault-tolerant distributed algorithms. We consider *reliable broadcast* and *fast decision* for a consensus algorithm. In the case of reliable broadcast we consider different fault models, namely, crashes [7] and Byzantine faults [29], as well as a hybrid fault model [30] with both, Byzantine and crash failures. For fast decision, we consider the one-step consensus algorithm BOSCO for Byzantine faults [28].

Reliable broadcast for crash and/or Byzantine failures. Figure 6 shows a sketch threshold automaton of a reliable broadcast that should tolerate $f_c \leq t_c$ crash and $f_b \leq t_b$ Byzantine faults under the resilience condition $n > 3t_b + 2t_c$. For our experiments under simpler failure models — only Byzantine and crash faults — we use the sketch threshold automata from Figures 2 and 5. However, the same thresholds can be obtained by setting $t_c = f_c = 0$ and $t_b = f_b = 0$ in the automaton from Figure 6, respectively. In Figure 2, we do not need a dedicated crash state, as we only model correct processes explicitly, while Byzantine faults are modeled via the guards (cf. Example 1). The automaton from Figure 5 can be obtained from Figure 6 by removing the location ℓ_{SE}.

[2] Theorem 7 and Corollary 8 in Appendix B consider floor and ceiling functions. Our benchmarks do not make use of such thresholds.

[3] See `http://forsyte.at/opodis17-artifact/` for detailed instructions on using the tool.

Table 1 Synthesized solutions for reliable broadcast that tolerates: crashes (Figure 5), Byzantine faults (Figure 2), and Byzantine & crash faults (Figure 6). We used the laptop in the experiments.

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > t_c,\ t_b = 0$	U, C, R	1	*true*	1	12	6
$n > 3t_b,\ t_c = 0$	U, C, R	3	$n - 2t_b$ $t_b + 1$ $t_b + 1$	$n - t_b$ $2t_b + 1$ $n - t_b$	31	16
$n \geq 3t_b,\ t_c = 0$	U, C, R	None	—	—	25	7
$n > 3t_b + 2t_c$	U, C, R	3	$n - 2t_b - 2t_c$ $t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c$ $n - t_b - t_c$	34	50
$n \geq 3t_b + 2t_c$	U, C, R	None	—	—	21	12
$n > 3t_b + t_c$	U, C, R	None	—	—	29	24

The algorithms we consider are the core of broadcasting algorithms, and establish agreement on whether to accept the message by the broadcaster. Similar to Example 1, processes start in locations ℓ_1 and ℓ_0, which capture that the process has received and has not received a message by the broadcaster, respectively. A correctly designed algorithm should satisfy the following properties [29]:

(U) *Unforgeability:* If no correct process starts in ℓ_1, then no correct process ever enters ℓ_{AC}.

(C) *Correctness:* If all correct processes start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

(R) *Relay:* Whenever a correct process enters ℓ_{AC}, all correct processes eventually enter ℓ_{AC}.

In the following discussion we use Figure 6 as example. We have to sketch the guards ϕ_{CR}, ϕ_{0toSE}, and ϕ_{AC}. At most f_c processes can move to the crashed state ℓ_{CR}. The algorithm designer does not have control over the crashes, and thus we fix the guard ϕ_{CR} to be $\mathsf{nc} < f_c$: The shared variable nc maintains the actual number of crashes (initially zero), which is used only to model crashes and thus cannot be used in guards other than ϕ_{CR}. To properly model that a processes can crash during a "send to all" operation (*non-clean crash*), we introduce two shared variables: the variable echos stores the number of echo messages that are sent by the correct processes (some of them may crash later), and the variable $\mathsf{echosCF}$ stores the number of echo messages that are sent by the correct processes and the faulty processes when crashing. Hence, the action *send* increases both echos and $\mathsf{echosCF}$, whereas the action *sendF* increases only $\mathsf{echosCF}$.

We define the thresholds τ_{0toSE} and τ_{AC} as $(\mathbf{?}_a^{\mathsf{SE}} \cdot n + \mathbf{?}_b^{\mathsf{SE}} \cdot t_b + \mathbf{?}_c^{\mathsf{SE}} \cdot t_c + \mathbf{?}_d^{\mathsf{SE}})$ and $(\mathbf{?}_a^{\mathsf{AC}} \cdot n + \mathbf{?}_b^{\mathsf{AC}} \cdot t_b + \mathbf{?}_c^{\mathsf{AC}} \cdot t_c + \mathbf{?}_d^{\mathsf{AC}})$ respectively. Hence, ϕ_{0toSE} and ϕ_{AC} are defined as $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{0toSE}}$ and $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{AC}}$. As discussed in the introduction, we add f_b to $\mathsf{echosCF}$ to reflect that the correct processes may — although do not have to — receive messages from Byzantine processes. For *reliable communication*, we have to enforce:

Every correct process eventually receives at least echos *messages.* (RelComm)

As threshold automata do not explicitly store the number of received messages, we transform (RelComm) into a fairness constraint, which forces processes to eventually leave a location if the messages by correct processes alone enable a guard of an edge that is outgoing from this location. That is, *there is a time after which the following holds forever:*

$$\boldsymbol{\kappa}[\ell_1] = 0 \wedge (\mathsf{echos} < \tau_{\mathsf{0toSE}} \vee \boldsymbol{\kappa}[\ell_0] = 0) \wedge (\mathsf{echos} < \tau_{\mathsf{AC}} \vee (\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 0)). \quad \text{(Fair)}$$

Table 2 Synthesized solutions for variations of reliable broadcast and specifications (X)–(Z).

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > 3t_b,\ t_c = 0$	X, C, R	None	—	—	15	2
$n > 3t_b + 2,\ t_c = 0$	X, C, R	3	$n - 2t_b$ $t_b + 3$ $t_b + 3$	$n - t_b$ $2t_b + 3$ $n - t_b$	35	12
$n > 3t_b,\ t_c = 0$	Y, C, R	None	—	—	28	6
$n > 4t_b,\ t_c = 0$	Y, C, R	3	$n - 2t_b$ $2t_b + 1$ $2t_b + 1$	$n - t_b$ $3t_b + 1$ $n - t_b$	33	12
$n > 3t_b + 2t_c$	U, Z, R	2	$t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c + 1$	41	31

Table 1 summarizes the experimental results for reliable broadcast, when looking for integer solutions only. The cases $t_b = 0$ and $t_c = 0$ correspond to the algorithms that tolerate only crashes (Figure 5) and only Byzantine faults (Figure 2) respectively. For these cases, we obtained the solutions known from the literature [29, 7] and some variations. Moreover, when the resilience condition is changed from $n > 3t_b$ to $n \geq 3t_b$, our tool reports no solution, which also complies with the literature [29]. In the case of f_c crashes and f_b Byzantine faults, the tool reports three solutions. Moreover, when we tried to relax the resilience condition to $n \geq 3t_b + 2t_c$ and $n > 3t_b + t_c$, the tool reported that there is no solution, as expected.

Variations of the specification. Our logic allows us to easily change the specifications. For instance, we can replace the precondition of unforgeability "if *no* correct process starts in ℓ_1" by giving an upper bound (number or parameter) on correct processes starting in ℓ_1 that still prevents entering ℓ_{AC}, in specifications (X) and (Y). We also changed the precondition of correctnesss "if *all* correct processes start in ℓ_1" in specification (Z):

(X) If *at most two* correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.
(Y) If *at most* t_b correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.
(Z) If *at least* $t_b + t_c + 1$ non-Byzantine processes (correct or crash faulty) start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

Interestingly, we obtain new distributed computing problems that put quantitative conditions on the initial state. These specifications are related to the specifications of condition-based consensus [27]. Our tool automatically generates solutions, or shows their absence in the case resilience conditions are too strong. Table 2 summarizes these results.

Byzantine one-step consensus. Figure 7 shows a sketch threshold automaton of a one-step Byzantine consensus algorithm that should tolerate $f \leq t$ Byzantine faults under the assumption $n > 3t$. It is a formalization of the BOSCO algorithm [28]. The purpose of the algorithm is to quickly reach consensus if (a) $n > 5t$ and $f = 0$, or (b) $n > 7t$. In this encoding, correct processes make a "fast" decision on 0 or 1 by going in the locations ℓ_{D0} and ℓ_{D1}, respectively. When neither (a) nor (b) holds, the processes precompute their votes in the first step and then go to the locations ℓ_{U0} and ℓ_{U1}, from which an *underlying consensus* algorithm is taking over. In this sense, BOSCO can be seen as an asynchronous preprocessing step for general consensus algorithms, and the properties given below contain preconditions for calling consensus in a safe way (see Fast Agreement below). Every run of a synthesized threshold automaton must satisfy the following properties (for $i \in \{0, 1\}$ and $j = 1 - i$):

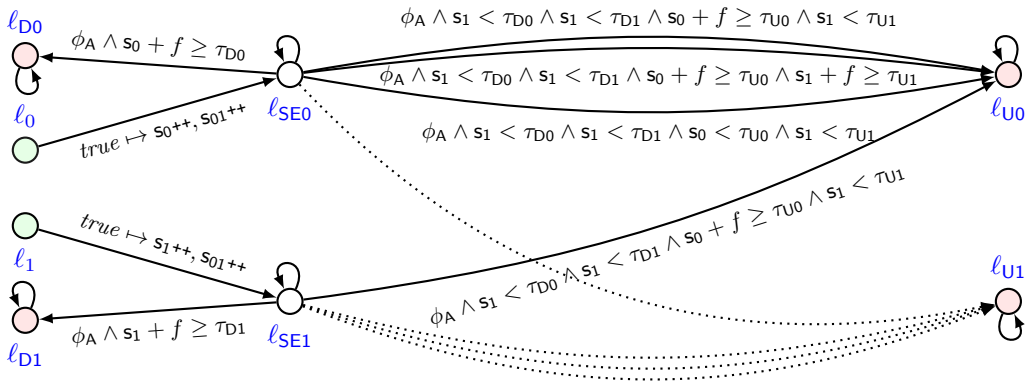


Figure 7 A sketch threshold automaton for one-step Byzantine consensus. Labels of dashed edges are omitted; they can be obtained from the respective solid edges by swapping 0 and 1.

(A) *Fast agreement [28, Lemmas 3–4]:* Condition $\boldsymbol{\kappa}[\ell_{\mathsf{Di}}] \neq 0$ implies $\boldsymbol{\kappa}[\ell_{\mathsf{Dj}}] = \boldsymbol{\kappa}[\ell_{\mathsf{Uj}}] = 0$.

(O) *One step:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it always holds that $\boldsymbol{\kappa}[\ell_{\mathsf{Dj}}] = 0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{U0}}] = \boldsymbol{\kappa}[\ell_{\mathsf{U1}}] = 0$. That is, the underlying consensus is never called.

(F) *Fast termination:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

(T) *Termination:* It eventually holds that $\boldsymbol{\kappa}[\ell_0] = \boldsymbol{\kappa}[\ell_1] = 0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{SE0}}] = \boldsymbol{\kappa}[\ell_{\mathsf{SE1}}] = 0$.

We define thresholds τ_{A}, τ_{D0}, τ_{D1}, τ_{U0}, τ_{U1} as $\mathbf{?}^x_a \cdot n + \mathbf{?}^x_b \cdot t + \mathbf{?}^x_c$ for $x \in \{\mathsf{A}, \mathsf{D0}, \mathsf{D1}, \mathsf{U0}, \mathsf{U1}\}$. Then, the guard ϕ_{A} is defined as: $s_{01} + f \geq \tau_{\mathsf{A}}$. Interestingly, the thresholds appear in different roles in the guards, e.g., $s_0 + f \geq \tau_{\mathsf{D0}}$ and $s_0 < \tau_{\mathsf{D0}}$. These cases correspond to BOSCO's decisions on how many messages *have been received* and how many messages *have not been received* "modulo Byzantine faults."

As with reliable broadcast, we model reliable communication with the following fairness constraint: For $i \in \{0, 1\}$, *from some point on*, the following holds: $\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0 \wedge (s_{01} < \tau_{\mathsf{A}} \vee s_i < \tau_{\mathsf{Di}} \vee \boldsymbol{\kappa}[\ell_{\mathsf{SEi}}] = 0)$.

We bound *denominators of rationals with two* and use the sanity box provided by Corollary 5. To reduce the search space, we assume that the guards for 0 and 1 are *symmetric*, that is $\mathbf{?}^{\mathsf{D0}}_a = \mathbf{?}^{\mathsf{D1}}_a$ and $\mathbf{?}^{\mathsf{U0}}_a = \mathbf{?}^{\mathsf{U1}}_a$. Still, BOSCO is a challenging benchmark for verification [19] and synthesis. Since the verification procedure from Section 3 independently checks schemas with SMT, we *parallelized* schema checking with OpenMPI, and ran the experiments at Vienna Scientific Cluster (VSC-3) using 8–128 cores; Table 3 summarizes the results. The tool has found four solutions for the guards: $\tau_{\mathsf{A}} = n - t\ [-\frac{1}{2}]$, $\tau_{\mathsf{D0}} = \tau_{\mathsf{D1}} = \frac{n+3t+1}{2}$, and $\tau_{\mathsf{U0}} = \tau_{\mathsf{U1}} = \frac{n-t}{2}\ [+\frac{1}{2}]$. In addition to the guards from [28], the tool also reported that one can add or subtract ½ from several guards. Figure 8 demonstrates that increasing the number of cores above 64 slows down synthesis times for this benchmark.

Variations of the BOSCO specifications. We relaxed the precondition for fast termination:

(U) If $n \geq 5t \wedge f = 0$ and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

(V) If $n \geq 7t$ and initially $\boldsymbol{\kappa}[\ell_{\mathsf{j}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

As can be seen from Table 3, specifications (U) and (V) have no solutions.

Table 3 Experiments for one-step Byzantine consensus for $n > 3t$ running the parallel verifier at VSC-3.

Specs	Nr. of solutions	Calls to verifier	Nr. of cores	Time min.
AOFT	4	516	128	39
AOFT	4	432	96	25
AOFT	4	425	64	24
AOFT	4	502	16	44
AOFT	4	440	8	51
AOUT	0	376	8	40
AOVT	0	337	8	33

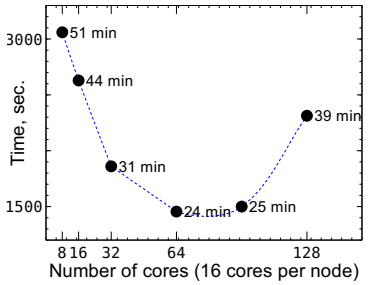


Figure 8 Synthesis times for BOSCO at Vienna Scientific Cluster (VSC-3).

6 Discussions

The classic approach to establish correctness of a distributed algorithm is to start with a system model, a specification, and pseudo code, all given in natural language and mathematical definitions, and then write a manual proof that confirms that "all fits together." Manual correctness proofs mix code inspection, system assumptions, and reasoning about events in the past and the future. Slight modifications to the system assumptions or the code require us to redo the proof. Thus, the proofs often just establish correctness of the algorithm, rather than deriving details of the algorithm — like the threshold guards — from the system assumption or the specification.

We introduced an automated method that synthesizes a correct distributed algorithm from the specifications and the basic assumptions. Our tool computes threshold expressions from the resilience condition and the specification, by learning the constraints that are derived from counterexamples. Learning dramatically reduces the number of verifier calls. In case of BOSCO, the sanity box contains 2^{36} vectors of unknowns, which makes exhaustive search impractical, while our technique only needs to check approximately 500 vectors.

In addition to synthesizing known algorithms from the literature, we considered several modified specifications. For some of them, our tool synthesizes thresholds, while for others it reports that no algorithm of a specific form exists. The latter results are indeed impossibility results (lower bounds on the fraction of correct processes) for fixed sketch threshold automata.

To ensure termination of the synthesis loop, we restrict the search space, and thus the class of algorithms for which the impossibility result formally applies. First, while we restrict the search to sane guards, the same synthesis loop can also be used to synthesize other guards. However, in order to ensure termination, a suitable characterization of sought-after guards should be provided by the user. Second, for reliable broadcast we consider only threshold guards with integer coefficients that can express thresholds like $n - t$ or $2t + 1$. For BOSCO, we only allow division by 2, and can express thresholds like $\frac{n}{2}$ or $\frac{n-t}{2}$. While from a theoretical viewpoint these restrictions limit the scope of our results, we are not aware of a distributed algorithm where processes wait for messages from, say, $\frac{n}{7}$ or $\frac{n}{1000}$ processes. To strengthen our completeness claim, we would need to formally explain why only small denominators are used in fault-tolerant distributed algorithms.

References

1 Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek

Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.

2 K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.

3 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

4 Benjamin Bisping, Paul-David Brodmann, Tim Jungnickel, Christina Rickmann, Henning Seidler, Anke Stüber, Arno Wilhelm-Weidner, Kirstin Peters, and Uwe Nestmann. A constructive proof for FLP. *Archive of Formal Proofs*, 2016.

5 Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and Byzantine-resilient distributed systems. In *CAV*, volume 9779 of *LNCS*, pages 157–176, 2016.

6 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

7 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

8 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

9 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, volume 6173 of *LNCS*, pages 142–148, 2010.

10 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

11 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. Springer Berlin Heidelberg, 2008.

12 Danny Dolev, Keijo Heljanko, Matti Järvisalo, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, and Siert Wieringa. Synchronous counting and computational algorithm design. *J. Comput. Syst. Sci.*, 82(2):310–332, 2016.

13 Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.

14 Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *TAAS*, 10(3):21:1–21:26, 2015.

15 Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *FORTE*, volume 9688 of *LNCS*, pages 124–141, 2016.

16 Adrià Gascón and Ashish Tiwari. A synthesized algorithm for interactive consistency. In *NFM*, volume 8430 of *LNCS*, pages 270–284. Springer, 2014.

17 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, 2017.

18 Swen Jacobs and Roderick Bloem. Parameterized synthesis. *LMCS*, 10(1:12), 2014.

19 Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734, 2017.

20 Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

21 Igor Konnov, Josef Widder, Francesco Spegni, and Luca Spalazzi. Accuracy of message counting abstraction in fault-tolerant distributed algorithms. In *VMCAI*, pages 347–366, 2017.

22 Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

23 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370, 2016.

24 Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

25 Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV*, pages 217–237, 2017.

26 Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *SSS*, volume 8756 of *LNCS*, pages 237–251, 2014.

27 Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

28 Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

29 T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

30 Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Dist. Comp.*, 20(2):115–140, 2007.

31 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

APPENDIX

A Detailed Proofs

In order to prove Theorem 4, we first prove mathematical background of it, i.e., Lemma 6.

▶ **Lemma 6.** *Fix a $k \in \mathbb{N}$, and for every $i \in \{1, \ldots, k\}$ fix $\delta_i > 0$. Let $a, b_1, \ldots, b_k, c$ be rationals for which the following holds: for every $n, t_1, \ldots, t_k \in \mathbb{N}$ such that $n > \sum_{i=1}^{k} \delta_i t_i \geq 0$, it holds that $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$. Then it is the case that*

$$0 \leq a \leq 1, \tag{7}$$

$$-\delta_i - 1 < b_i < \delta_i + 1, \text{ for all } i = 1, \ldots, k \tag{8}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{9}$$

Proof. Let $\mathbf{P}_{RC}$ be the set of all tuples $(n, t_1, \ldots, t_k) \in \mathbb{N}^{k+1}$ that satisfy $n > \sum_{i=1}^{k} \delta_i t_i \geq 0$. Thus, we assume that for $a, b_1, \ldots, b_k, c \in \mathbb{Q}$ the following holds:

$$0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{10}$$

We show that if any of the conditions (7)–(9) is violated, we obtain a contradiction by finding $(n^0, t_1^0, \ldots, t_k^0) \in \mathbf{P}_{RC}$ such that $0 \leq an^0 + \sum_{i=1}^{k} b_i t_i^0 + c \leq n^0$ does not hold.

Proof of (7). Let us first show that $0 \leq a \leq 1$.

Assume by contradiction that $a > 1$. From (10) we know that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^{k} b_i t_i + c$, that is, $(1-a)n \geq \sum_{i=1}^{k} b_i t_i + c$. Since $1 - a < 0$, we obtain

$$n \leq \frac{\sum_{i=1}^{k} b_i t_i + c}{1-a}, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{11}$$

Consider any tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ where $n^0 > \max\left\{\sum_{i=1}^{k} \delta_i t_i^0, \frac{\sum_{i=1}^{k} b_i t_i^0 + c}{1-a}\right\}$. By construction, we obtain: (i) the tuple is in $\mathbf{P}_{RC}$ because $n^0 > \sum_{i=1}^{k} \delta_i t_i^0$, and (ii) we have $n^0 > \frac{\sum_{i=1}^{k} b_i t_i^0 + c}{1-a}$, such that we arrive at the required contradiction to (11).

Assume now that $a < 0$. Again from (10) we have that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{i=1}^{k} b_i t_i + c \geq 0$, or in other words $an \geq -\sum_{i=1}^{k} b_i t_i - c$. As $a < 0$, this means that

$$n \leq \frac{-\sum_{i=1}^{k} b_i t_i - c}{a}, \text{ for every } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{12}$$

Consider a tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ with $n^0 > \max\left\{\sum_{i=1}^{k} \delta_i t_i^0, \frac{-\sum_{i=1}^{k} b_i t_i^0 - c}{a}\right\}$. By construction it holds that $n^0 > \sum_{i=1}^{k} \delta_i t_i^0$, and thus the tuple is in $\mathbf{P}_{RC}$. Also by construction it holds that $n^0 > \frac{-\sum_{i=1}^{k} b_i t_i^0 - c}{a}$ which is a contradiction with (12).

Proof of (8). Let us now prove that $-\delta_i - 1 < b_i < \delta_i + 1$, for an arbitrary $i \in \{1, \ldots, k\}$.

Assume by contradiction that $b_i \geq \delta_i + 1$. Recall from (10) that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{j=1}^{k} b_j t_j + c \leq n$, or in other words $(1-a)n \geq \sum_{j=1}^{k} b_j t_j + c$. Since $a \in [0, 1]$, then $(1-a)n \leq n$, for every $n \geq 0$. Since $b_i \geq \delta_i + 1$, and $t_i \geq 0$, it holds that $b_i t_i \geq (\delta_i + 1)t_i$. Thus, we have that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds that

$$n \geq (1-a)n \geq \sum_{j=1}^{k} b_j t_j + c \geq (\delta_i + 1)t_i + \sum_{j \neq i} b_j t_j + c.$$

In other words, we have that

$$(n - \delta_i t_i) - \sum_{j \neq i} b_j t_j - c \geq t_i, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{13}$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ such that $t_i^0 = \max\{1, \sum_{j \neq i}(\delta_j - b_j) - c + 2\}$, $t_j^0 = 1$ for $j \neq i$, and $n^0 = \sum_{j=1}^{k} \delta_j t_j^0 + 1 = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{j=1}^{k} \delta_j t_j^0$. Let us check the inequality from (13). By construction we have $(n^0 - \delta_i t_i^0) - \sum_{j \neq i} b_j t_j^0 - c = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 - \sum_{j \neq i} b_j - c$, that is, $\sum_{j \neq i}(\delta_j - b_j) - c + 1$, which is strictly smaller than t_i^0 by construction. Thus, we obtained a contradiction with (13).

Let us now assume $b_i \leq -\delta_i - 1$. Recall from (10) that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{j=1}^{k} b_j t_j + c$. Since $a \in [0, 1]$, for every $n \in \mathbb{N}$ holds $an \leq n$, and since $b_i \leq -\delta_i - 1$, we have $b_i t_i \leq -\delta_i t_i - t_i$, for every $t_i \geq 0$. Thus, for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ we have

$$0 \leq an + \sum_{j=1}^{k} b_j t_j + c \leq n + (-\delta_i t_i - t_i) + \sum_{j \neq i} b_j t_j + c.$$

In other words, we have that

$$t_i \leq (n - \delta_i t_i) + \sum_{j \neq i} b_j t_j + c, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{14}$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ where $t_i^0 = \max\{\sum_{j\neq i}(\delta_j + b_j) + c + 2, 1\}$, $t_j^0 = 1$, for every $j \neq i$, and $n^0 = \sum_{j=1}^k \delta_j t_j^0 + 1 = \sum_{j\neq i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$, since $n^0 > \sum_{i=1}^k \delta_i t_i^0$. Let us check the inequality from (14). By construction we have $(n^0 - \delta_i t_i^0) + \sum_{j\neq i} b_j t_j^0 + c = \sum_{j\neq i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 + \sum_{j\neq i} b_j + c$, that is, $\sum_{j\neq i}(\delta_j + b_j) + c + 1$, which is strictly smaller than t_i^0 by construction. This gives us a contradiction with (14).

Proof of (9). And finally, let us prove that $-2\sum_{i=1}^k \delta_i - k - 1 \leq c \leq 2\sum_{i=1}^k \delta_i + k + 1$.

Assume by contradiction that $c > 2\sum_{i=1}^k \delta_i + k + 1$. Recall that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^k b_i t_i + c$, by (10). Since $a \geq 0$, $b_i > -\delta_i - 1$, for every $i = 1, \ldots, k$, and $c > 2\sum_{i=1}^k \delta_i + k + 1$, then we have that

$$n \geq an + \sum_{i=1}^k b_i t_i + c > \sum_{i=1}^k (-\delta_i - 1) t_i + 2\sum_{i=1}^k \delta_i + k + 1, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \quad (15)$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0)$ where $t_1^0 = \ldots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. The tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $\sum_{i=1}^k (-\delta_i - 1) t_i^0 + 2\sum_{i=1}^k \delta_i + k + 1 = \sum_{i=1}^k \delta_i + 1 = n^0$, which is a contradiction with (15).

Assume by contradiction that $c < -2\sum_{i=1}^k \delta_i - k - 1$. Recall that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{i=1}^k b_i t_i + c$, by (10). Since $a \leq 1$, $b_i < \delta_i + 1$, for every $i = 1, \ldots, k$, and $c < -2\sum_{i=1}^k \delta_i - k - 1$, then we have that

$$0 \leq an + \sum_{i=1}^k b_i t_i + c < n + \sum_{i=1}^k (\delta_i + 1) t_i - 2\sum_{i=1}^k \delta_i - k - 1, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \quad (16)$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0)$ where $t_1^0 = \ldots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $n^0 + \sum_{i=1}^k (\delta_i + 1) t_i^0 - 2\sum_{i=1}^k \delta_i - k - 1 = 0$, which is a contradiction with (16). ◀

Proof of Theorem 4. As the given guard is sane for the resilience condition, the number compared against a shared variable should have a value from 0 to n. For every tuple $(n, t_1, \ldots, t_k)$ of parameter values satisfying the resilience condition, it should hold that $0 \leq an + \sum_{i=1}^k b_i t_i + c \leq n$. We may thus apply Lemma 6 and the theorem follows. ◀

Proof of Corollary 5. Using the fact that $x \leq \frac{\tilde{d}}{D} \leq y$ implies that $Dx \leq \tilde{d} \leq Dy$, for a $D \in \mathbb{N}$, this corollary follows directly from Theorem 4. ◀

B Thresholds with floor and ceiling functions

The following theorem considers threshold guards that use the ceiling or the floor function. It uses the same reasoning as in Theorem 4, combined with the properties of these functions. Namely, for every $x \in \mathbb{R}$ it holds that $x \leq \lceil x \rceil < x + 1$ and $x - 1 < \lfloor x \rfloor \leq x$.

▶ **Theorem 7.** *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^k \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \ldots, k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(an + (b_1 t_1 + \ldots + b_k t_k) + c\right) \quad \text{or} \quad x < f\left(an + (b_1 t_1 + \ldots + b_k t_k) + c\right),$$

where $x \in \Gamma$ is a shared variable, $a, b_1, \ldots, b_k, c \in \mathbb{Q}$ are rationals, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds that

$$0 \leq a \leq 1, \tag{17}$$

$$-\delta_i - 1 < b_i < \delta_i + 1, \text{ for all } i = 1, \ldots, k, \tag{18}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 2 \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k, \text{ if } f \text{ is floor, or} \tag{19}$$

$$-2(\delta_1 + \ldots + \delta_k) - k \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k + 2, \text{ if } f \text{ is ceiling.} \tag{20}$$

Proof sketch. The proof largely follows the arguments of the proof of Lemma 6 with fixed denominators as in Corollary 5. The only remaining issue is that instead of constraints of the form $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$, that are considered in Lemma 6, here we have to argue about constraints of the form $0 \leq f\left(an + \sum_{i=1}^{k} b_i t_i + c\right) \leq n$, where f is the ceiling or the floor function.

Let us first discuss the case when f is the ceiling function. As for every $x \in \mathbb{R}$ holds that $x \leq \lceil x \rceil < x + 1$, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil < an + (b_1 t_1 + \ldots + b_k t_k) + c + 1.$$

Still, as the guard is sane, we have that $0 \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil \leq n$. Combining these two constraints, we obtain that

$$0 < an + (b_1 t_1 + \ldots + b_k t_k) + (c + 1) \quad \text{and} \quad an + (b_1 t_1 + \ldots + b_k t_k) + c \leq n.$$

With these constraints, we can derive a contradiction following the proof of Lemma 6.

Similarly, if f is the floor function, we use the fact that for every $x \in \mathbb{R}$ holds that $x - 1 < \lfloor x \rfloor \leq x$. Therefore, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c - 1 < \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq an + (b_1 t_1 + \ldots + b_k t_k) + c.$$

As $0 \leq \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq n$, we obtain that

$$0 \leq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{and} \quad an + (b_1 t_1 + \ldots + b_k t_k) + (c - 1) < n.$$

And again, the rest of the proof follows the line of the proof of Lemma 6. ◀

If coefficients in guards have a fixed denominator, we can obtain intervals for numerators as a direct consequence of Theorem 7.

▶ **Corollary 8.** *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \ldots, k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(\frac{\tilde{a}}{D} n + \sum_{i=1}^{k} \frac{\tilde{b_i}}{D} t_i + \frac{\tilde{c}}{D}\right) \quad \text{or} \quad x < f\left(\frac{\tilde{a}}{D} n + \sum_{i=1}^{k} \frac{\tilde{b_i}}{D} t_i + \frac{\tilde{c}}{D}\right),$$

where $x \in \Gamma$ is a shared variable, $\tilde{a}, \tilde{b_1}, \ldots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$ are integers, $D \in \mathbb{N}$, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds

$$0 \leq a \leq D, \tag{21}$$

$$D(-\delta_i - 1) < b_i < D(\delta_i + 1), \text{ for all } i = 1, \ldots, k, \tag{22}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k - 2) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k), \text{ if } f \text{ is floor, or} \tag{23}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k + 2), \text{ if } f \text{ is ceiling.} \tag{24}$$