20th International Conference on Principles of Distributed Systems

OPODIS 2016, December 13–16, 2016, Madrid, Spain

Edited by
Panagiota Fatourou
Ernesto Jiménez
Fernando Pedone

Editors

Panagiota Fatourou	Ernesto Jiménez	Fernando Pedone
FORTH ICS & Depart. of Comp. Sci., Univ. of Crete	Technical University of Madrid (UPM)	University of Lugano (USI)
Crete	Madrid	Lugano
Greece	Spain	Switzerland
faturu@ics.forth.gr	ernes@etsisi.upm.es	fernando.pedone@usi.ch

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 978-3-95977-031-6

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/978-3-95977-031-6.

Publication date
April, 2017

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.2016.0

ISBN 978-3-95977-031-6 **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

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Session 3: Population Protocols, Agents, and Robots

Session 4: Languages and Systems

Session 5: Combinatorics and Randomization

Session 6: Replication and Consensus

Preface

The papers in this volume were presented at the 20th International Conference on Principles of Distributed Systems (OPODIS 2016), held on December 13–16, 2016, in Spain. The conference was organized by the Technical University of Madrid (UPM) and took place in the Fundación Gómez Pardo in Madrid.

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 84 submissions and each submission was reviewed by at least three members of the Program Committee with the help of external reviewers, with most papers receiving four reviews or more. This year, the papers of the Program Committee members were reviewed by at least five members of the Program Committee. Overall, the quality of the submissions was very high. Out of the 84 submissions, 31 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 procces was done using Easychair.

The Best Paper Award was given to David Yu Cheng Chan, Vassos Hadzilacos and Sam Toueg for the paper "Bounded Disagreement". Moreover, some of the papers were selected

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

by the Program Committee for a special issue to appear in Theoretical Computer Science, Elsevier.

This year OPODIS had four distinguished invited keynote speakers: Christian Cachin (IBM Research Zurich, Switzerland), Faith Ellen (University of Toronto, Canada), Nir Shavit (MIT, USA), and Willy Zwaenepoel (EPFL, 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 and their availability during the physical Program Committee meeting which was held at the University of Lugano on October 20–21, 2016 (with the support of FORTH ICS that provided the teleconference infrastructure). 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, consisting of: Ángel Álvarez, Pilar Manzano, Isabel Muñoz (Technical University of Madrid), Antonio Fernández Anta, Nicolas Nicolaou (Institute IMDEA Networks), Sergio Arévalo (Technical University of Madrid) as local chair, José Luis López-Presa (Technical University of Madrid) as publicity chair, and Andrés Sevilla (Technical University of Madrid) as responsible for the website. 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.

December 2016

Panagiota Fatourou, FORTH ICS & Department of Computer Science, Univ. of Crete
Ernesto Jiménez Merino, Technical University of Madrid (UPM)
Fernando Pedone, University of Lugano

Committees

General Chair

Ernesto Jiménez, Technical University of Madrid (UPM), Spain

Program Chairs

Panagiota Fatourou, FORTH ICS & Department of Computer Science, University of Crete, Greece
Fernando Pedone, University of Lugano (USI), Switzerland

Program Committee

Yehuda Afek, Tel-Aviv University, Israel
Christian Cachin, IBM Research, Zurich, Switzerland
Marco Canini, KAUST, Saudi Arabia
Shantanu Das, Aix-Marseille University, France
Carole Delporte, Université Paris Diderot – Paris 7, France
Fernando Dotti, PUC-RS, Brazil
Faith Ellen, University of Toronto, Canada
Pascal Felber, Université de Neuchâtel, Switzerland
Pierre Fraigniaud, Université Paris Diderot – Paris 7, France
Cyril Gavoille, Université de Bordeaux, France
Rachid Guerraoui, EPFL, Switzerland
Maurice Herlihy, Brown University, USA
Rüdiger Kapitza, TU Braunschweig, Germany
Parisa Marandi, Microsoft Research, UK
Euripides Markou, University of Thessaly, Greece
Alessia Milani, University of Bordeaux, France
Gilles Muller, INRIA, France
Roberto Palmieri, Virginia Tech, USA
Marta Patiño-Martínez, Technical University of Madrid, Spain
David Peleg, Weizmann Institute of Science, Israel
Sebastiano Peluso, Virginia Tech, USA
Maria Potop Butucaru, Université Paris-VI Pierre-et-Marie-Curie, France
Nuno Preguiça, Universidade Nova de Lisboa, Portugal
Luis Rodrigues, Technical University of Lisbon, Portugal
Eric Ruppert, York University, Canada
Mark Shapiro, INRIA & Université Paris-VI Pierre-et-Marie-Curie, France
Liuba Shrira, Brandeis University, USA
Robert Soule, University of Lugano, Switzerland
Roman Vitenberg, University of Oslo, Norway
Spyros Voulgaris, VU University Amsterdam, Netherlands

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Steering Committee

Marcos Aguilera, VMware Research Group, USA
Christian Cachin, IBM Research, Zurich, Switzerland
Alessia Milani, University of Bordeaux, France
Maria Potop-Butucaru, Université Paris-VI Pierre-et-Marie-Curie, France
Giuseppe Prencipe, Università di Pisa , Italy
Etienne Rivière, University of Neuchâtel, Switzerland
Marc Shapiro, INRIA & Université Paris-VI Pierre-et-Marie-Curie, France
Sebastien Tixeuil (steering committee chair), IUF & Université Pierre et Marie Curie – Paris 6, France

Organization Committee

Ángel Álvarez,Technical University of Madrid (UPM), Spain
Sergio Arévalo (organization chair),Technical University of Madrid (UPM), Spain
Antonio Fernández-Anta, Institute IMDEA Networks, Spain
José Luis López-Presa (publicity chair), Technical University of Madrid (UPM), Spain
Pilar Manzano, Technical University of Madrid (UPM), Spain
Isabel Muñoz, Technical University of Madrid (UPM), Spain
Nicolas Nicolaou, Institute IMDEA Networks, Spain
Andrés Sevilla, Technical University of Madrid (UPM), Spain

Additional Reviewers

Abhinav Aggarwal
Dan Alistarh
Sergio Arevalo
Jordi Arjona
James Aspnes
Valter Balegas
Evangelos Bampas
Leonid Barenboim
Samuel Benz
Lelia Blin
Marcus Brandenburger
Stefan Brenner
Janna Burman
Claire Capdevielle
Antonio Carzaniga
Keren Censor-Hillel
Jeremie Chalopin
Chen Chen
Giorgos Christou
Andrea Clementi
Tyler Crain
Jurek Czyzowicz
Eliyahu Dain
Antonella Del Pozzo
Giuseppe Antonio Di Luna
Stefan Dobrev
Swan Dubois
Yuval Emek
Hugues Fauconnier
Antonio Fernandez Anta
Paola Flocchini
Leszek Gasieniec
Rati Gelashvili
Chryssis Georgiou
Seth Gilbert
Emmanuel Godard
Vincent Gramoli
Sandeep Hans
Ahmed Hassan
Danny Hendler
Damien Imbs
Michiko Inoue
Leander Jehl
Colette Johnen
Nikolaos Kallimanis
Eleni Kanellou
Christina Karousatou
David Kirkpatrick
Ralf Klasing
Peter Kling
Rastislav Kralovic
Evangelos Kranakis
Shay Kutten
Petr Kuznetsov
Arnaud Labourel
Mikel Larrea
Jonas Lefevre
Joao Leitao
Jonathan Lejeune
Christoph Lenzen
Albert Linde
Victor Luchangco
Khaled Maamra
Marc X. Makkes
Arthur Martens
Pierre Matri
Alex Matveev
Hein Meling
Hugues Mercier
Avery Miller
Calvin Newport
Fukuhito Ooshita
Ami Paz
Andrzej Pelc
Katerina Potika
Charlie Rackoff
Tomasz Radzik
Leila Ribeiro
Peter Robinson
Pierre-Louis Roman
Jared Saia
Dimitris Sakavalas
Vikram Saraph
Guillaume Scerri
Nicolas Schabanel
Valerio Schiavoni
Michele Scquizzato
Pierre Sens
Lili Su
Gadi Taubenfeld
Alejandro Z. Tomsic
Corentin Travers
Jacopo Urbani
Przemyslaw Uznanski
Nitin Vaidya
Xenofon Vasilakos
Valerio Vianello
Nico Weichbrodt
Jennifer Welch
Wenbo Xu
Akka Zemmari
Kaiwen Zhang

List of Authors

Hamid Arabnejad
IC4, Dublin City University, Dublin, Ireland
hamid.arabnejad@dcu.ie

James Aspnes
Yale University, USA
james.aspnes@gmail.com

Aras Atalar
Chalmers University of Technology, Sweden
aaras@chalmers.se

Jorge Barbosa
LIACC, Faculdade de Engenharia, Universidade do Porto, Portugal
jbarbosa@fe.up.pt

Joffroy Beauquier
LRI, Paris-South University, France
joffroy.beauquier@lri.fr

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

Annalisa De Bonis
Università di Salerno, Italy
debonis@dia.unisa.it

Janna Burman
LRI, Paris-South University, France
janna.burman@lri.fr

Christian Cachin
IBM Research – Zürich, Switzerland
cca@zurich.ibm.com

Arnaud Casteigts
LaBRI, University of Bordeaux, France
acasteig@labri.fr

David Yu Cheng Chan
Department of Computer Science, University of Toronto, Canada
davidchan@cs.toronto.edu

Yen-Jung Chang
Department of Electrical and Computer Engineering, University of Texas at Austin, USA
cyenjung@ece.utexas.edu

Tian Ze Chen
Department of Computer Science, University of Toronto, Canada
tianze.chen@mail.utoronto.ca

Johanne Cohen
LRI-CNRS, Université Paris-Sud, Université Paris Saclay, France
johanne.cohen@lri.fr

Carole Delporte-Gallet
IRIF, Université Paris-Diderot, France
cd@liafa.univ-paris-diderot.fr

John Derrick
Department of Computing, University of Sheffield, UK
j.derrick@sheffield.ac.uk

Stéphane Devismes
Université Grenoble Alpes, Grenoble, France
stephane.devismes@imag.fr

Simon Doherty
Department of Computing, University of Sheffield, UK
s.doherty@sheffield.ac.uk

Brijesh Dongol
Department of Computer Science, Brunel University, UK
brijesh.dongol@brunel.ac.uk

Hugues Fauconnier
IRIF, Université Paris-Diderot, France
hf@liafa.univ-paris-diderot.fr

Antonio Fernández Anta
IMDEA Networks Institute, Spain
antonio.fernandez@imdea.org

Eli Gafni
UCLA, USA
eli@ucla.edu

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Vijay Garg
Department of Electrical and Computer Engineering, University of Texas at Austin, USA
garg@ece.utexas.edu

Leszek Gasieniec
Department of Computer Science, University of Liverpool, UK
l.a.gasieniec@liverpool.ac.uk

Mozhgan Ghasemzadeh
LIACC, Faculdade de Engenharia, Universidade do Porto, Portugal
ghasemzadeh@fe.up.pt

Emmanuel Godard
Aix Marseille Univ, CNRS, LIF, France
emmanuel.godard@lif.univ-mrs.fr

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

Theophanis Hadjistasi
University of Connecticut, USA
theophanis.hadjistasi@uconn.edu

Vassos Hadzilacos
Department of Computer Science, University of Toronto, Canada
vassos@cs.toronto.edu

David Hamilton
Department of Computer Science, University of Liverpool, UK
d.d.hamilton@liverpool.ac.uk

Meng He
Faculty of Computer Science, Dalhousie University, Canada
mhe@cs.dal.ca

Yuan He
UCLA, USA
yuan.he@ucla.edu

Heidi Howard
University of Cambridge Computing Laboratory, Cambridge, UK
heidi.howard@cl.cam.ac.uk

David Ilcinkas
Univ. Bordeaux & CNRS, LaBRI, France
david.ilcinkas@labri.fr

Leander Jehl
University of Stavanger, Norway
leander.jehl@uis.no

Colette Johnen
Univ. Bordeaux & CNRS, LaBRI, France
johnen@labri.fr

Idit Keidar
Department of Electrical Engineering, Technion, Israel
idish@ee.technion.ac.il

Pankaj Khanchandani
ETH Zurich, Switzerland
kpankaj@ethz.ch

Kishori Konwar
Department of EECS, MIT, USA
kishori@csail.mit.edu

Petr Kuznetsov
Télécom ParisTech, France
petr.kuznetsov@telecom-paristech.fr

Kim Larsen
Aalborg University, Denmark
kgl@cs.aau.dk

Mengdu Li
Dark Matter LLC, Canada
meng.du.li@dal.ca

Nancy Lynch
Department of EECS, MIT, USA
lynch@csail.mit.edu

Khaled Maâmra
LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, France
khaled.maamra@uvsq.fr

Dahlia Malkhi
VMware, Palo Alto, USA
dahliamalkhi@gmail.com

George Manoussakis
LRI-CNRS, Université Paris-Sud, Université Paris Saclay, France
george.manoussakis@lri.fr

Russell Martin
Department of Computer Science, University of Liverpool, UK
russell.martin@liverpool.ac.uk

Fabien Mathieu
Nokia Bell Labs, France
fabien.mathieu@nokia-bell-labs.com

Alexandre Maurer
EPFL, Switzerland
alexandre.maurer@epfl.ch

Muriel Médard
Department of EECS, MIT, USA
medard@mit.edu

Hein Meling
University of Stavanger, Norway
hein.meling@uis.no

Ricardo Mendes
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
rmendes@lasige.di.fc.ul.pt

Yves Métivier
LaBRI, University of Bordeaux, France
metivier@labri.fr

Nicolas Nicolaou
IMDEA Networks Institute, Spain
nicolas.nicolaou@imdea.org

Stavros Nikolaou
Department of Computer Science, Cornell University, USA
snikolaou@cs.cornell.edu

Tiago Oliveira
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
toliveira@lasige.di.fc.ul.pt

Sriram Pemmaraju
Department of Computer Science, University of Iowa, USA
sriram-pemmaraju@uiowa.edu

Eloi Perdereau
Aix Marseille Univ, CNRS, LIF, France
eloi.perdereau@lif.univ-mrs.fr

Laurence Pilard
LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, France
laurence.pilard@uvsq.fr

Narayana Prakash
Department of EECS, MIT, USA
prakashn@mit.edu

Ammar Qadri
University of Toronto, Canada
ammar.qadri@mail.utoronto.ca

Paul Renaud-Goud
Toulouse Institute of Computer Science Research, France
prenaud@irit.fr

Robbert Van Renesse
Department of Computer Science, Cornell University, USA
rvr@cs.cornell.edu

Talal Riaz
Department of Computer Science, University of Iowa, Iowa City, USA
talal-riaz@uiowa.edu

Thibault Rieutord
Télécom ParisTech, France
thibault.rieutord@telecom-paristech.fr

John Michael Robson
LaBRI, University of Bordeaux, France
robson@labri.fr

Gerhard Schellhorn
Universität Augsburg, Institut für Informatik, Germany
schellhorn@informatik.uni-augsburg.de

Simon Schubert
IBM Research – Zürich, Switzerland
sis@zurich.ibm.com

Stefan Schmid
Aalborg University, Denmark
schmiste@cs.aau.dk

Devan Sohier
Université de Versailles, LI-PaRAD, France
devan.sohier@uvsq.fr

Alexander Spiegelman
Department of Electrical Engineering, Technion, Israel
sashas@tx.technion.ac.il

Paul Spirakis
Department of Computer Science, University of Liverpool, UK
p.spirakis@liverpool.ac.uk

Grzegorz Stachowiak
Instytut Informatyki, Uniwersytet Wrocławski, Poland
gst@cs.uni.wroc.pl

Sam Toueg
Department of Computer Science, University of Toronto, Canada
sam@cs.toronto.edu

Philippas Tsigas
Chalmers University of Technology, Sweden
tsigas@chalmers.se

Nitin H. Vaidya
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, USA
nhv@illinois.edu

Marko Vukolić
IBM Research – Zürich, Switzerland
mvu@zurich.ibm.com

Roger Wattenhofer
ETH Zürich, Switzerland
wattenhofer@ethz.ch

Heike Wehrheim
Universität Paderborn, Institut für Informatik, Germany
wehrheim@upb.de

Yuanhao Wei
Department of Computer Science, University of Toronto, Canada
yuanhao.wei@mail.utoronto.ca

Zhuolun Xiang
Department of Computer Science, University of Illinois at Urbana-Champaign, USA
xiangzl@illinois.edu

Bingtian Xue
Aalborg University, Denmark
bingt@cs.aau.dk

Akka Zemmari
LaBRI, University of Bordeaux, France
zemmari@labri.fr

High Throughput Connectomics

Nir Shavit

Massachusetts Institute of Technology (MIT), Cambridge, MA, USA
shanir@csail.mit.edu

Abstract

Connectomics is an emerging field of neurobiology that uses cutting edge machine learning and image processing to extract brain connectivity graphs from electron microscopy images. It has long been assumed that the processing of connectomics data will require mass storage and farms of CPUs and GPUs and will take months if not years. This talk will discuss the feasibility of designing a high-throughput connectomics-on-demand system that runs on a multicore machine with less than 100 cores and extracts connectomes at the terabyte per hour pace of modern electron microscopes. Building this system required solving algorithmic and performance engineering issues related to scaling machine learning on multicore architectures, and may have important lessons for other problem spaces in the natural sciences, where until now large distributed server or GPU farms seemed to be the only way to go.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Machine learning, multicore architectures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.1

Category Keynote

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Blockchain – From the Anarchy of Cryptocurrencies to the Enterprise

Christian Cachin

IBM Research, Zürich, Switzerland
cca@zurich.ibm.com

Abstract

A blockchain is a public ledger for recording transactions, maintained by many nodes without central authority through a distributed cryptographic protocol. All nodes validate the information to be appended to the blockchain, and a consensus protocol ensures that the nodes agree on a unique order in which entries are appended. Distributed protocols tolerating faults and adversarial attacks, coupled with cryptographic tools are needed for this. The recent interest in blockchains has revived research on consensus protocols, ranging from the proof-of-work method in Bitcoin's "mining" protocol to classical Byzantine agreement. Going far beyond its use in cryptocurrencies, blockchain is today viewed as a promising technology to simplify trusted exchanges of data and goods among companies. In this context, the Hyperledger Project has been established in early 2016 as an industry-wide collaborative effort to develop an open-source blockchain. This talk will present an overview of blockchain concepts, cryptographic building blocks and consensus mechanisms. It will also introduce Hyperledger Fabric, an implementation of blockchain technology intended for enterprise applications. Being one of the key partners in the Hyperledger Project, IBM is actively involved in the development of this blockchain platform.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases consensus, cryptographic, distributed protocols

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.2

Category Keynote

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Really Big Data: Analytics on Graphs with Trillions of Edges

Willy Zwaenepoel

School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

Big graphs occur naturally in many applications, most obviously in social networks, but also in many other areas such as biology and forensics. Current approaches to processing large graphs use either supercomputers or very large clusters. In both cases the entire graph must reside in memory before it can be processed. We are pursuing an alternative approach, processing graphs from secondary storage. While this comes with a performance penalty, it makes analytics on very large graphs feasible on a small number of commodity machines. We have developed two systems, one for a single machine and one for a cluster of machines. X-Stream, the single machine solution, aims to make all secondary storage access sequential. It uses two techniques to achieve this goal, edge-centric processing and streaming partitions. Chaos, the cluster solution, starts from the observation that there is little benefit to locality when accessing data from secondary storage over a high-speed network. As a result, Chaos spreads graph data uniformly randomly over storage devices, and uses randomized access to achieve I/O balance. Chaos furthermore uses work stealing to achieve computational load balance. By using these techniques, it avoids the need for expensive partitioning during pre-processing, while still achieving good scaling behavior. With Chaos we have been able to process an 8-trillion-edge graph on 32 machines, a new milestone for graph size on a small cluster. I will describe both systems and their performance on a number of benchmarks and in comparison to state-of-the-art alternatives. This is joint work with Laurent Bindschaedler (EPFL), Jasmina Malicevic (EPFL) and Amitabha Roy (Intel Labs).

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases large graphs

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.3

Category Keynote

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Participating Sets, Simulations, and the Consensus Hierarchy

Faith Ellen

Department of Computer Science, University of Toronto, Canada
faith@cs.toronto.edu

Abstract

The participating set problem can be solved in an asynchronous system using only registers. I will gently explain this problem and its solution, followed by a new extension, called consistent ordered partition. Next, I will present a wait-free simulation by $f + 1$ processes of any set-consensus algorithm that tolerates f faults. I will also describe how to extend this simulation using consistent ordered partition. Finally, I will discuss how this extension can be used to prove that, within every level $m > 1$ of the consensus hierarchy, there is an infinite sequence of increasingly more powerful deterministic objects.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Consensus, shared-memory systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.4

Category Keynote

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 4; pp. 4:1–4:1

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Bounded Disagreement

David Yu Cheng Chan[1], Vassos Hadzilacos[2], and Sam Toueg[3]

1 **Department of Computer Science, University of Toronto, Ontario, Canada**
davidchan@cs.toronto.edu
2 **Department of Computer Science, University of Toronto, Ontario, Canada**
vassos@cs.toronto.edu
3 **Department of Computer Science, University of Toronto, Ontario, Canada**
sam@cs.toronto.edu

Abstract

A well-known generalization of the consensus problem, namely, *set agreement (SA)*, limits the number of distinct *decision values* that processes decide. In some settings, it may be more important to limit the number of "disagreers". Thus, we introduce another natural generalization of the consensus problem, namely, *bounded disagreement (BD)*, which limits the number of *processes* that decide differently from the plurality. More precisely, in a system with n processes, the (n, ℓ)-BD task has the following requirement: there is a value v such that at most ℓ processes (the disagreers) decide a value other than v. Despite their apparent similarities, the results described below show that bounded disagreement, consensus, and set agreement are in fact fundamentally different problems.

We investigate the relationship between bounded disagreement, consensus, and set agreement. In particular, we determine the *consensus number* [16] for every instance of the BD task. We also determine values of n, ℓ, m, and k such that the (n, ℓ)-BD task can solve the (m, k)-SA task (where m processes can decide at most k distinct values). Using our results and a previously-known impossibility result for set agreement [8], we prove that for all $n \geq 2$, there is a BD task (and a corresponding BD object) that has consensus number n but can *not* be solved using n-consensus and registers. Prior to our paper, the only objects known to have this unusual characteristic for $n \geq 2$ (which shows that the consensus number of an object is not sufficient to fully capture its power) were artificial objects crafted solely for the purpose of exhibiting this behaviour [2, 18].

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Consensus, Set Agreement, Asynchronous System, Distributed Algorithms, Shared Memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.5

1 Introduction

Reaching agreement among processes is a fundamental problem in fault-tolerant distributed computing. The strongest and most-studied formulation of such a problem is the *consensus* task [12], where all non-faulty processes must decide on *one* of their input values. Another well-known agreement problem is a generalization of consensus, namely, the *set agreement (SA)*[1] task [7], which limits the number of distinct *decision values* that processes decide. In this paper we introduce and study another natural generalization of consensus, one that

[1] Also known as *set consensus*.

limits the number of *processes* that decide differently from the plurality; we call this the *bounded disagreement (BD)* task.

In the consensus, SA and BD tasks, there are n processes, each process has an input value (its *proposal*) and must output a value (its *decision*) such that every correct process eventually outputs a decision value (*Termination*) and each decision is one of the proposals (*Validity*). Additionally, the (n, k)-SA task must also satisfy the following property:

- **k-SA**: There exist at most k distinct decision values.

For $k = 1$, the (n, k)-SA task is simply the n-consensus task; in this case, there is only one decision value and hence no disagreers. For $k = 2$, however, there are up to two decision values, but the number of disagreers (i.e., those deciding differently from the majority) can suddenly jump all to the way to $\lfloor n/2 \rfloor$. In some settings, it may be more important to limit the number of processes that disagree than limiting the number of different decision values. Thus we introduce the (n, ℓ)-BD task that replaces the k-SA property with the following one:

- **ℓ-BD**: There is a value v such that at most ℓ processes (the *disagreers*) decide a value other than v.

For $\ell = 0$, the (n, ℓ)-BD task is just the n-consensus task. Furthermore, having at most ℓ disagreers implies having at most $\ell + 1$ distinct decision values, so the (n, ℓ)-BD task is at least as strong as the $(n, \ell + 1)$-SA task.

In this paper, we study the relationship between the consensus, set agreement and bounded disagreement tasks, and show that despite their similarity, these tasks are fundamentally different. To do so, we first determine values of n, ℓ, m and k such that the (n, ℓ)-BD task and registers can solve the (m, k)-SA task. We then determine the *consensus number* [16] of every instance of the BD task: specifically, we prove that the (n, ℓ)-BD task has consensus number $\max(n - 2\ell, 1)$.[2] Using the above results and a previously-known impossibility result for set agreement [8], we prove that BD tasks have the following uncommon property: for all $j \geq 2$, there is a BD task, namely the $(9j, 4j)$-BD task, that has consensus number j but *cannot* be solved using j-consensus objects and registers. These results imply that there are infinitely many instances of the BD task that are not equivalent to any consensus task or any set agreement task (because for $k \geq 2$ all (m, k)-SA tasks have consensus number 1).[3]

We also define the (n, ℓ)-BD *object* that solves the (n, ℓ)-BD *task*, as follows: intuitively, processes can "propose" any value to this object and it responds to the first n such proposals with values that satisfy the Validity and ℓ-BD properties; all subsequent proposals return arbitrary responses. It turns out that the above results about BD tasks also apply to the corresponding BD objects. In particular, for all $j \geq 2$, the $(9j, 4j)$-BD object has consensus number j, but cannot be implemented using j-consensus and registers. The existence of objects with this unusual property has been the subject of much research [2, 3, 4, 5, 9, 10, 11, 15, 18]. Prior to this paper, however, the only objects known to have this property for $j \geq 2$ were artificial objects crafted solely for the purpose of exhibiting this behaviour [2, 18]. Unlike these objects, however, our BD objects and tasks are natural generalizations of consensus. The BD objects and those in [18] are non-deterministic, while the objects in [2] are deterministic.

[2] The consensus number of a task T is the maximum number of processes for which consensus can be solved using any solution of T and registers.

[3] We say that two tasks are equivalent if any solution to one task, together with registers, can be used to solve the other task. Similarly, we say that two objects are equivalent if instances of either object, together with registers, can be used to implement the other.

Roadmap. In Section 2, we determine values of ℓ, m, k, and n such that the (m, ℓ)-BD task can solve the (n, k)-SA task. In Section 3, we define BD objects that can be used to solve BD tasks, and, in Section 4, we investigate what instances of consensus can *not* be solved using these BD objects. Together, these results allow us to determine the consensus number of every instance of the BD task and BD object. In Section 5, we prove that for all $n \geq 2$, there are instances of the BD task and BD object that have consensus number n, yet cannot be implemented using n-consensus and registers. In Section 6, we conclude with a brief discussion on bounded disagreement and related problems.

2 Using BD Tasks to Solve SA Tasks

Recall that with the SA and BD tasks, each process starts with an input value (its *proposal*) and must output a value (its *decision*), subject to certain restrictions. A solution to such a task is a collection of protocols $\mathcal{P}$, one for each process to execute in order to produce decisions that respect these restrictions.

In this section, we show that for all $k \geq 1$, $n \geq 1$, $\ell \geq 0$ and $m \geq \max(n+\ell+\lfloor \ell/k \rfloor, 2\ell+2)$, the (m, ℓ)-BD task and registers can solve the (n, k)-SA task.[4] For $k = 1$ and $n = m - 2\ell$, this says that the (m, ℓ)-BD task and registers can solve the $(m - 2\ell, 1)$-SA task, i.e., the $(m - 2\ell)$-consensus task.

We prove this result in two steps: we first assume that processes have access to a *Fetch&Increment* object, and then we remove this assumption. Intuitively, a Fetch&Increment object $\mathcal{F}$ is just a counter: each operation on $\mathcal{F}$ returns the current value of the counter and increments it by one [5].

Multisets notation. It is convenient to use a *multiset* to store the outputs of a BD task. Given any multiset M, the *mode of* M, denoted *mode*(M), is the value that appears the most often in M (ties are broken arbitrarily), and $|M|$ is the total number of (non-necessarily distinct) elements in M. The $+$ operator denotes the *disjoint union* of two multisets. This is distinct from the normal union operator $\cup$ in that the elements are added regardless of whether they are distinct from other elements.

▶ **Theorem 1.** *For all $k \geq 1$, $n \geq 1$, $\ell \geq 0$ and $m \geq n + \ell + \lfloor \ell/k \rfloor$, the (m, ℓ)-BD task and registers, together with a Fetch&Increment object, can solve the (n, k)-SA task.*

Proof. Consider any $k \geq 1$, $n \geq 1$, $\ell \geq 0$ and $m \geq n + \ell + \lfloor \ell/k \rfloor$. We present an algorithm that solves the (n, k)-SA task using:

- $\mathcal{P}[1..m]$, a solution for the (m, ℓ)-BD task.
- $\mathcal{X}$, an n-process snapshot object where each field contains a multiset of integers; initially empty. (Note that snapshot objects can be implemented from registers [1].)
- $\mathcal{F}$, a Fetch&Increment object; initialized such that the first operation will have response 1.

Recall that an n-process snapshot object $\mathcal{X}$ is composed of n fields, and provides two types of operations, write and snapshot. A write operation, denoted by $\mathcal{X}[p]$.WRITE(v), replaces the value of field $\mathcal{X}[p]$ by v. A snapshot operation, denoted by $\mathcal{X}$.SNAPSHOT(), returns the values in all n fields at once.

Intuitively, each process p uses $\mathcal{F}$ to determine which protocol(s) in $\mathcal{P}[1..m]$ to execute, and uses the field $\mathcal{X}[p]$ to announce the multiset of all the outputs that p has received from

[4] That is, *any* solution to the (m, ℓ)-BD task can be used, together with registers, to derive a solution to the (n, k)-SA task.

Algorithm 1 Solving the (n, k)-SA task using an (m, ℓ)-BD task solution $\mathcal{P}[1..m]$, an atomic snapshot $\mathcal{X}$, and a Fetch&Increment object $\mathcal{F}$.

```
1: while TRUE do
2:     next ← F.FETCH&INCREMENT()
3:     if next ≤ m then
4:         received ← received + {P[next].execute(v_p)}
5:         X[p].WRITE(received)
6:     else
7:         snap ← X.SNAPSHOT()
8:         decide mode(snap)
9:         halt
```

executing protocols in $\mathcal{P}[1..m]$ so far. In an abuse of notation, we will also use $\mathcal{X}$ to denote the multiset formed by the disjoint union of the multisets in the n fields of $\mathcal{X}$.

In addition to these shared objects, each process p also uses the following local variables:

- *received*: a multiset used to store all outputs that p has received from executing protocols in $\mathcal{P}[1..m]$; initially empty.
- *snap*: a multiset used to store a snapshot of $\mathcal{X}$; initially empty.
- *next*: a positive integer.

We denote by var_p the local variable *var* of process p, and by v_p the input value of p. We now describe the algorithm that each process p executes to solve the (n, k)-SA task:

1. It executes a Fetch&Increment operation on $\mathcal{F}$ to get a positive integer i (line 2).
2. If $i \leq m$, it executes protocol $\mathcal{P}[i]$ with input value v_p, and adds the output to $\mathcal{X}[p]$ (lines 3 to 5).
3. If $i > m$, it takes a snapshot of $\mathcal{X}$, decides mode($\mathcal{X}$), and halts (lines 6 to 9).
4. It repeats from the first step.

For simplicity, let $\mathcal{P}[i].execute(v)$ denote the output from executing protocol $\mathcal{P}[i]$ with input value v. The pseudocode for each process p in shown in Algorithm 1.

▶ **Claim 2.** *Each protocol of $\mathcal{P}[1..m]$ is executed at most once.*

Proof. Follows immediately from the fact that each protocol $\mathcal{P}[i]$ for all $1 \leq i \leq m$ is executed only when the Fetch&Increment object $\mathcal{F}$ gives the response i to some process. ◀

Thus the outputs of $\mathcal{P}[1..m]$ must satisfy the requirements of the (m, ℓ)-BD task. Let $\texttt{Output}(\mathcal{P})$ be the multiset of all outputs from $\mathcal{P}[1..m]$ so far, and let $\texttt{Output}_p(\mathcal{P})$ be the multiset of all outputs from $\mathcal{P}[1..m]$ given to process p so far. To begin the analysis of this algorithm, first observe that the following invariant always holds:

▶ **Invariant 3.** *For each process p,*
(a) *$\mathcal{X}[p] \subseteq \texttt{Output}_p(\mathcal{P})$, and hence $\mathcal{X} \subseteq \texttt{Output}(\mathcal{P})$.*
(b) *$\mathcal{X}[p]$, and hence $\mathcal{X}$, is monotonically increasing.*

▶ **Claim 4** (Termination)**.** *Every correct process eventually decides exactly one value.*

Proof. Let p be a correct process. Since $next_p$ is derived from repeated operations on the Fetch&Increment object $\mathcal{F}$, it will eventually become greater than m. Then the conditional on line 3 will evaluate to false, causing p to decide exactly one value and halt (line 8). ◀

▶ **Claim 5** (Validity). *Every decision v was proposed by some process, i.e., $v = v_q$ for some process q.*

Proof. Suppose a process p decides a value v. Clearly, $v \in \mathcal{X}$. By Invariant 3, $\mathcal{X}$ is a subset of $\texttt{Output}(\mathcal{P})$, so v is an output from $\mathcal{P}[1..m]$. Since $\mathcal{P}[1..m]$ is a solution for the (m, ℓ)-BD task, by the Validity property of this task, v must be one of the input values for $\mathcal{P}[1..m]$. The claim follows from the fact that each process q only uses v_q as the input value for $\mathcal{P}[1..m]$. ◀

▶ **Claim 6.** *Consider any snapshot of $\mathcal{X}$ that contains at least $h \geq \ell$ elements. The mode of this snapshot appears at least $h - \ell$ times in $\texttt{Output}(\mathcal{P})$.*

Proof. Since $\mathcal{P}[1..m]$ is a solution for the (m, ℓ)-BD task, by the ℓ-BD property of this task, there is a value v such that at most ℓ of the outputs in $\texttt{Output}(\mathcal{P})$ are not v. By Invariant 3, every snapshot of $\mathcal{X}$ is a subset of $\texttt{Output}(\mathcal{P})$. Consequently, each snapshot of $\mathcal{X}$ contains at most ℓ outputs that are not v, and the rest of the outputs must equal v. In other words, if a snapshot of $\mathcal{X}$ contains at least h elements, then v appears at least $h - \ell$ times in this snapshot. Thus the mode of the snapshot of $\mathcal{X}$ appears at least $h - \ell$ times in the snapshot. By Invariant 3, the mode of the snapshot also appears at least $h - \ell$ times in $\texttt{Output}(\mathcal{P})$. ◀

▶ **Claim 7.** *If a process decides a value v, then v appears at least $\lfloor \ell/k \rfloor + 1$ times in $\texttt{Output}(\mathcal{P})$.*

Proof. Suppose a process p decides a value v. Then, by lines 8 and 7, v is the mode of a snapshot S of $\mathcal{X}$, and by the conditional on line 3, the snapshot S is taken after p get a value greater than m from $\mathcal{F}$. Since $\mathcal{F}$ is initialized such that the first operation on $\mathcal{F}$ gets a response of 1, at least $m + 1$ operations were performed on $\mathcal{F}$ before the snapshot S is taken.

Note that each time a process receives a response from $\mathcal{F}$ that is at most m, it cannot perform another operation on $\mathcal{F}$ before executing a protocol of $\mathcal{P}[1..m]$ and adding the output of that execution to the multiset stored by $\mathcal{X}$. Thus, since the n processes perform a total of at least $m + 1$ operations on $\mathcal{F}$ before the snapshot S is taken, at least $m - n + 1$ outputs were added to the multiset stored by $\mathcal{X}$ before the snapshot S is taken. Thus S contains at least $m - n + 1$ outputs.

Now, recall that $m \geq n + \ell + \lfloor \ell/k \rfloor$, so $m - n + 1 \geq \ell + \lfloor \ell/k \rfloor + 1$. Thus the snapshot S of $\mathcal{X}$ contains at least $\ell + \lfloor \ell/k \rfloor + 1$ outputs. By Claim 6, the mode v of S appears at least $\lfloor \ell/k \rfloor + 1$ times in $\texttt{Output}(\mathcal{P})$. ◀

▶ **Claim 8** (k-SA). *There are at most k distinct decision values.*

Proof. Assume, for contradiction, that there are at least $k + 1$ distinct decision values. By Claim 7, each decision value appears at least $\lfloor \ell/k \rfloor + 1$ times in $\texttt{Output}(\mathcal{P})$. In other words, $\texttt{Output}(\mathcal{P})$ contains at least $k+1$ distinct output values, each appearing at least $\lfloor \ell/k \rfloor + 1$ times. This implies that for *every* value v, at least $k(\lfloor \ell/k \rfloor + 1) > \ell$ of the outputs given by $\mathcal{P}[1..m]$ are not v. This violates the ℓ-BD property of the (m, ℓ)-BD task, contradicting the fact that $\mathcal{P}[1..m]$ solves the (m, ℓ)-BD task. ◀

Thus the Termination (Claim 4), Validity (Claim 5), and k-SA (Claim 8) properties of the (n, k)-SA task all hold. So Algorithm 1 solves the (n, k)-SA task. ◀

▶ **Theorem 9.** *For all $\ell \geq 0$ and $m \geq 2\ell + 2$, the (m, ℓ)-BD task and registers can solve the 2-consensus task.*

Algorithm 2 Solving consensus among 2 processes using an (m, ℓ)-BD task solution $\mathcal{P}[1..m]$ and an atomic snapshot $\mathcal{X}$.

```
1:  while TRUE do
2:      snap ← X.SNAPSHOT()
3:      if |snap| ≤ m − 2 then
4:          if p = 1 then
5:              received ← received + {P[|received| + 1].execute(v_p)}
6:          else
7:              received ← received + {P[m − |received|].execute(v_p)}
8:          X[p].WRITE(received)
9:      else
10:         decide mode(snap)
11:         halt
```

Proof. Consider any $\ell \geq 0$ and $m \geq 2\ell + 2$. We present a modification of Algorithm 1 that solves the 2-consensus task without using a Fetch&Increment object. In addition to a solution $\mathcal{P}[1..m]$ for the (m, ℓ)-BD task, the new algorithm uses only a (shared) 2-process snapshot object $\mathcal{X}$, and local variables *received* and *snap* as described in Algorithm 1.

Recall that the Fetch&Increment object $\mathcal{F}$ used in Algorithm 1 serves two roles:

- It ensures each protocol of $\mathcal{P}[1..m]$ is executed at most once.
- It tells processes when to take a snapshot of $\mathcal{X}$ and decide the mode.

Roughly speaking, processes can decide as soon as sufficiently many outputs have been written into $\mathcal{X}$. Thus, the second role can be replaced simply by having processes take a snapshot of $\mathcal{X}$ and counting the current number of outputs. For the first role, we can ensure that each protocol of $\mathcal{P}[1..m]$ is executed at most once by using the fact that there are only two processes involved in the 2-consensus task. Intuitively, this is achieved by starting the two processes at opposite ends of the range $[1..m]$, and having them sequentially execute the protocols of $\mathcal{P}[1..m]$ until they meet each other.

As before, let v_p denote p's input value. Let the two processes be numbered 1 and 2. We now describe the algorithm that each process p executes to solve the 2-consensus task:

1. It takes a snapshot S of $\mathcal{X}$.
2. If S contains more than $m - 2$ outputs, it decides the mode.
3. If p is process 1, it executes protocol $\mathcal{P}[i_1 + 1]$ with input value v_p, where i_1 is the number of protocols process 1 has executed.
4. If p is process 2, it executes protocol $\mathcal{P}[m - i_2]$ with input value v_p, where i_2 is the number of protocols process 2 has executed.
5. It adds the output to $\mathcal{X}[p]$.
6. It repeats from the first step.

The pseudocode for each process p in shown in Algorithm 2.

▶ **Claim 10.** *Each protocol of $\mathcal{P}[1..m]$ is executed at most once.*

Proof. Assume for contradiction, that for some $1 \leq i \leq m$, protocol $\mathcal{P}[i]$ is executed more than once. Observe that in Algorithm 2, this can only happen if both processes execute protocol $\mathcal{P}[i]$, since each process sequentially traverses the range $[1..m]$. This means both processes took a snapshot of $\mathcal{X}$ and found at most $m - 2$ outputs right before executing protocol $\mathcal{P}[i]$. However, in order for both processes to reach protocol $\mathcal{P}[i]$, they must first

execute all other protocols in $\mathcal{P}[1..m]$ and write their outputs into $\mathcal{X}$. Hence one of the two processes must have found more than $m-2$ outputs in its snapshot, a contradiction. ◀

Since each protocol is executed at most once, their outputs must satisfy the requirements of the (m,ℓ)-BD task. As before, let $\texttt{Output}(\mathcal{P})$ be the multiset of all outputs from $\mathcal{P}[1..m]$ so far, and let $\texttt{Output}_p(\mathcal{P})$ be the multiset of all outputs from $\mathcal{P}[1..m]$ given to process p so far. First, note that Invariant 3 still holds for this modified algorithm. Furthermore, Claim 5 and Claim 6 also hold here: their proofs are exactly as before. The remaining claims also have similar proofs:

▶ **Claim 11** (Termination). *Every correct process eventually decides exactly one value.*

Proof. Let p be a correct process. Observe that each time the conditional on line 3 evaluates to true, process p will execute a protocol of $\mathcal{P}[1..m]$, and write its output to $\mathcal{X}$, increasing the number of outputs in $\mathcal{X}$ by one. By Invariant 3, the number of outputs in $\mathcal{X}$ is monotonically increasing. Thus it will eventually become greater than $m-2$. Then the conditional on line 3 will evaluate to false, causing p to decide exactly one value and halt (lines 10). ◀

▶ **Claim 12.** *If a process decides a value v, then v appears at least $\ell+1$ times in $\texttt{Output}(\mathcal{P})$.*

Proof. Suppose a process p decides a value v. Then, by lines 10 and 2, v is the mode of a snapshot S of $\mathcal{X}$, and by the conditional on line 3, the snapshot S contains at least $m-1$ outputs. Now, recall that $m \geq 2\ell+2$, so $m-1 \geq 2\ell+1$. Thus the snapshot S of $\mathcal{X}$ contains at least $2\ell+1$ outputs. By Claim 6, the mode v of S appears at least $\ell+1$ times in $\texttt{Output}(\mathcal{P})$. ◀

▶ **Claim 13** (Agreement). *All decisions have the same value.*

Proof. Assume, for contradiction, that there are at least 2 distinct decision values. By Claim 12, each decision value appears at least $\ell+1$ times in $\texttt{Output}(\mathcal{P})$. Thus $\texttt{Output}(\mathcal{P})$ contains at least 2 distinct output values, each appearing at least $\ell+1$ times. This implies that for *every* value v, at least $\ell+1 > \ell$ of the outputs given by $\mathcal{P}[1..m]$ are not v. This violates the ℓ-BD property of the (m,ℓ)-BD task, contradicting the fact that $\mathcal{P}[1..m]$ solves the (m,ℓ)-BD task. ◀

Thus the Termination (Claim 11), Validity (Claim 5), and Agreement (Claim 13) properties of the 2-consensus task all hold. So Algorithm 2 solves the 2-consensus task. ◀

We can now prove the main result of this section:

▶ **Theorem 14.** *For all $k \geq 1$, $n \geq 1$, $\ell \geq 0$ and $m \geq \max(n+\ell+\lfloor \ell/k \rfloor, 2\ell+2)$, the (m,ℓ)-BD task and registers can solve the (n,k)-SA task.*

Proof. In previous work, Afek *et al.* [5] proved that (any solution to) the 2-consensus task, together with registers, can be used to implement a Fetch&Increment object for any number of processes. Thus from Theorem 9, for all $\ell \geq 0$ and $m \geq 2\ell+2$, the (m,ℓ)-BD task and registers can implement a Fetch&Increment object for any number of processes. The result now follows from Theorem 1. ◀

▶ **Corollary 15.** *For all $\ell \geq 0$ and $m > 2\ell$, the (m,ℓ)-BD task and registers can solve the $(m-2\ell)$-consensus task.*

Proof. Let $\ell \geq 0$. For $m = 2\ell+1$, the $(m-2\ell)$-consensus task is the trivial 1-consensus task. For $m \geq 2\ell+2$, by setting $k=1$ and $n = m-2\ell$ in Theorem 14, we get that the (m,ℓ)-BD task and registers can solve the $(m-2\ell,1)$-SA task, i.e., the $(m-2\ell)$-consensus task. ◀

3 BD Objects

Intuitively, with an (m, ℓ)-BD *object*, processes can propose any value to the object, and the object responds to the first m such proposals with values that satisfy the Validity and ℓ-BD properties; the $(m+1)$-th proposal permanently *upsets* the (m, ℓ)-BD object, causing it to nondeterministically respond with an arbitrary value to this proposal and all the subsequent ones. Thus, as long as the BD object is not upset, every response is one of the proposals (Validity) and satisfies the following property:

- **ℓ-BD**: There exists a value v such that at most ℓ responses are not v.

Note that m processes can trivially use a single (m, ℓ)-BD object to solve the (m, ℓ)-BD task: each process just proposes its input value to the object and decides the object's response. However, since the (m, ℓ)-BD object gets upset if more than m proposal operations are applied, it cannot be used in this trivial way by $m' > m$ processes to solve the (m', ℓ)-BD task.

We now give a more precise specification of the (m, ℓ)-BD object. A linearizable shared memory object is specified by a tuple $(OP, RES, Q, s_0, \delta)$, where OP is a set of operations, RES is a set of response values, Q is a set of states, $s_0 \in Q$ is the initial state, and $\delta \subseteq Q \times OP \times Q \times RES$ is a state transition relation such that:

- A tuple $(s, op, s', res) \in \delta$ means that if the object is in state s when operation $op \in OP$ is invoked, then the object can change its state to s' and return res as the response.
- δ is total: For every reachable state $s \in Q$ and every operation $op \in OP$, there exists at least one $s' \in Q$ and $res \in RES$ such that $(s, op, s', res) \in \delta$.

The state of an (m, ℓ)-BD object $\mathcal{D}$ is either a pair (S_{OP}, M_{RES}), where S_{OP} is the *set* of all the values proposed to $\mathcal{D}$ and M_{RES} is the *multiset* of all responses given by $\mathcal{D}$, or the upset state $\perp$. Note that the state of $\mathcal{D}$ does *not* record information about the *order* of operations and responses. If the state of $\mathcal{D}$ is a pair (S_{OP}, M_{RES}), we denote by $|M_{RES}|$ the size of M_{RES}; note that $|M_{RES}|$ is also the number of operations performed on $\mathcal{D}$ so far. We say that the state (S_{OP}, M_{RES}) is *full* if $|M_{RES}| = m$, since the next operation performed on $\mathcal{D}$ will cause $\mathcal{D}$ to permanently enter the upset state $\perp$.

For simplicity, we define the set of operations to be $\mathbb{Z}$ (the set of integers), where operation $i \in \mathbb{Z}$ represents proposing the value i. So for all $\ell \geq 1$ and $m > \ell$, we define the (m, ℓ)-BD object as the tuple $(OP, RES, Q, s_0, \delta)$ such that:

- $OP = \mathbb{Z}$.
- $RES = \mathbb{Z}$.
- The set of states Q consists of:
  - Every pair (S_{OP}, M_{RES}), where S_{OP} is a set of integers and M_{RES} is a multiset of integers such that $|M_{RES}| \leq m$. If $|M_{RES}| = m$, the state is called *full*.
  - The upset state $\perp$.
- $s_0 = (\varnothing, \varnothing)$.
- The state transition relation δ contains an element (s, op, s', res) if and only if one of the following two conditions holds:
  - s and s' are states (S_{OP}, M_{RES}) and (S'_{OP}, M'_{RES}) such that all of the following hold:
    - s is not full.
    - $S'_{OP} = S_{OP} \cup \{op\}$.
    - $M'_{RES} = M_{RES} + \{res\}$.
    - **Validity:** Each element in M'_{RES} is in S'_{OP}.
    - **ℓ-BD:** There exists a value v such that at most ℓ elements in M'_{RES} are not v.
  - s is either a full state or the upset state $\perp$, and s' is the upset state $\perp$.

From these definitions, we can make a few observations. First, the validity property implies:

▶ **Observation 16.** *The response to the first operation performed on a BD object is the operation's own proposal value.*

Next, the upset state $\perp$ works as intended:

▶ **Observation 17.** *A BD object that is in either a full state or the upset state $\perp$ can nondeterministically respond with any value to all future operations.*

Furthermore, the validity and ℓ-BD properties hold as long as the object is not upset:

▶ **Observation 18.** *If at most m operations are applied to an (m, ℓ)-BD object $\mathcal{D}$, then each response given by $\mathcal{D}$ is a value proposed to $\mathcal{D}$, and there is a value v such that at most ℓ of the responses given by $\mathcal{D}$ are not v.*

This immediately implies:

▶ **Observation 19.** *The (m, ℓ)-BD task can be solved with a single (m, ℓ)-BD object.*

From the above observation, it is clear that Theorem 14 and Corollary 15 also apply for BD objects.

4 Unsolvability of $(m - 2\ell + 1)$-Consensus by (m, ℓ)-BD Objects

We now show that (m, ℓ)-BD objects and registers *cannot* solve the $(m - 2\ell + 1)$-consensus task. This, together with Corollary 15, allows us to determine the consensus number of every instance of the BD task and BD object.

▶ **Theorem 20.** *For all $\ell \geq 0$ and $m > 2\ell$, (m, ℓ)-BD objects and registers cannot solve the $(m - 2\ell + 1)$-consensus task.*

Proof. Consider any $\ell \geq 0$ and $m > 2\ell$. For the special case of $\ell = 0$, observe that the (m, ℓ)-BD object is equivalent to an m-consensus object. Thus the theorem immediately follows from the fact that m-consensus objects and registers cannot solve the $(m+1)$-consensus task. Hence it suffices to consider the case where $\ell \geq 1$.

For $\ell \geq 1$, we prove a stronger result, namely, that (m, ℓ)-BD objects and registers cannot solve the *binary*[5] $(m - 2\ell + 1)$-consensus task, even when the (m, ℓ)-BD objects are further strengthened by restricting their nondeterministic behavior as follows: As long as it is not upset, each (m, ℓ)-BD object outputs at most two distinct response values.

Assume, for contradiction, that there is an $\ell \geq 1$, an $m > 2\ell$ and a wait-free algorithm that solves binary consensus among $m - 2\ell + 1 \geq 2$ processes using only registers and the strengthened (m, ℓ)-BD objects. It suffices to prove the existence of an infinite execution of this algorithm where processes never decide.

The proof uses the bivalency technique introduced by Fischer *et al.* [12]. We assume the reader is familiar with bivalency proof terminology such as *configuration*, *bivalent*, and 0-*valent* [12, 16]. In the following, we omit the proofs of Claims 21 to 24 since they are standard (see [16]).

▶ **Claim 21.** *The algorithm has an initial bivalent configuration C_{init}.*

[5] In the binary consensus task, every proposal value is restricted to being either 0 or 1.

▶ **Claim 22.** *There is a bivalent configuration C_{bi}, reachable from C_{init}, such that if any process takes a step, the resulting configuration is univalent.*

▶ **Claim 23.** *At C_{bi}, every process is about to perform an operation on the same object $\mathcal{D}$.*

▶ **Claim 24.** *$\mathcal{D}$ is not a register.*

Since the algorithm only uses registers and (m, ℓ)-BD objects,

▶ **Claim 25.** *$\mathcal{D}$ is an (m, ℓ)-BD object.*

▶ **Claim 26.** *There exists a pair of distinct processes p_0 and p_1, and a pair of (not necessarily distinct) values v_0 and v_1, such that starting from C_{bi},*

- *p_0 can take a step with response v_0, and the resulting configuration C_0 is 0-valent.*
- *p_1 can take a step with response v_1, and the resulting configuration C_1 is 1-valent.*

Proof. Follows immediately from Claim 22 since there are at least 2 processes. ◀

Given a configuration C such that $\mathcal{D}$ is not upset in C, we denote by $\mathcal{D}.M_{RES}(C)$ the multiset M_{RES} in the state of $\mathcal{D}$ in C.

▶ **Claim 27.** *$\mathcal{D}$ is not upset in C_{bi} and $|\mathcal{D}.M_{RES}(C_{bi})| < 2\ell$.*

Proof. Assume, for contradiction, that either $\mathcal{D}$ is upset in C_{bi} or $|\mathcal{D}.M_{RES}(C_{bi})| \geq 2\ell$. By Claim 23, every process was about to apply an operation on $\mathcal{D}$ at C_{bi}, and so at both C_0 and C_1, every process other than p_0 and p_1 will perform an operation on $\mathcal{D}$ as its next step. Thus consider the following configurations:

- A 0-valent configuration C_0' reached from C_0 by letting every process other than p_0 and p_1 (if any) take a step in an arbitrary order (while p_0 and p_1 take no steps).
- A 1-valent configuration C_1' reached from C_1 by letting every process other than p_0 and p_1 (if any) take a step in an arbitrary order (while p_0 and p_1 take no steps).

There are two cases:

1. $\mathcal{D}$ is upset in C_{bi}. Clearly, $\mathcal{D}$ is also upset in both C_0' and C_1'.
2. $|\mathcal{D}.M_{RES}(C_{bi})| \geq 2\ell$. Since there are $m - 2\ell + 1$ processes, by Claim 23, in both C_0' and C_1', the number of operations performed on $\mathcal{D}$ is $|\mathcal{D}.M_{RES}(C_{bi})| + (m - 2\ell + 1) - 1 \geq m$. Thus in both C_0' and C_1', $\mathcal{D}$ is in either a full state or the upset state $\perp$.

So in all cases, $\mathcal{D}$ is in either a full state or the upset state $\perp$, in both C_0' and C_1'. By Observation 17, $\mathcal{D}$ is henceforth allowed to nondeterministically return any response to all subsequent operations. Thus consider the following configurations:

- The 0-valent configuration C_0'' reached from C_0' by letting p_1 take a step with response v_1.
- The 1-valent configuration C_1'' reached from C_1' by letting p_0 take a step with response v_0.

Since p_0 received v_0 as the response from $\mathcal{D}$ in both C_0'' and C_1'', the state of p_0 is the same in C_0'' as in C_1''. Furthermore, the state of every object is the same in C_0'' as in C_1'': $\mathcal{D}$ is in the upset state $\perp$, and all other objects have not changed their state since C_{bi}. Consequently, if all processes other than p_0 crash, p_0 will not be able to distinguish between the 0-valent configuration C_0'' and the 1-valent configuration C_1'', so if p_0 runs solo starting from configurations C_0'' and C_1'', it decides the same value in both runs – a contradiction. ◀

▶ **Claim 28.** $|\mathcal{D}.M_{RES}(C_{bi})| \geq 1$.

Proof. Assume, for contradiction, that $|\mathcal{D}.M_{RES}(C_{bi})| = 0$. In other words, $\mathcal{D}$ is in the initial state in C_{bi}. By Observation 16, v_0 is the proposal value of the operation that p_0 is about to invoke on $\mathcal{D}$ at configuration C_{bi}, and similarly v_1 is the proposal value of the operation that p_1 is about to invoke on $\mathcal{D}$ at configuration C_{bi}. We claim that:

- Starting from C_0, p_1 can take a step with response v_1, and the resulting configuration C'_0 is 0-valent.
- Starting from C_1, p_0 can take a step with response v_0, and the resulting configuration C'_1 is 1-valent.

To see why, recall that $\mathcal{D}$ is a strengthened (m, ℓ)-BD object where $\ell \geq 1$. In both C'_0 and C'_1, the number of distinct response values from $\mathcal{D}$ is at most two, and the number of responses different from v_0 is at most 1. Thus the above steps required to reach C'_0 and C'_1 are possible.

Finally, recall that the state of $\mathcal{D}$ does not record information about the order of operations and responses: it records only the set of proposals and the multiset of responses so far. Consequently, the state of $\mathcal{D}$ is the same in C'_0 as in C'_1. Furthermore, since p_0 received v_0 as the response from $\mathcal{D}$ in both C'_0 and C'_1, the state of p_0 is the same in C'_0 as in C'_1. Similarly, the state of p_1 is the same in C'_0 as in C'_1. Finally, observe that all other objects and processes have not changed their state since C_{bi}. Thus their states are also the same in C'_0 as in C'_1. Therefore $C'_0 = C'_1$, contradicting the fact that C'_0 is 0-valent and C'_1 is 1-valent. ◀

By Claim 27 and Claim 28, $\mathcal{D}$ is not upset in C_{bi} and $1 \leq |\mathcal{D}.M_{RES}(C_{bi})| < 2\ell$. Let v_{mode} denote the mode of $\mathcal{D}.M_{RES}(C_{bi})$ (ties can be decided arbitrarily).

▶ **Claim 29.** *At most $\ell - 1$ of the responses in $\mathcal{D}.M_{RES}(C_{bi})$ are not v_{mode}.*

Proof. By Claim 27, $\mathcal{D}$ is not upset at C_{bi} and $\mathcal{D}.M_{RES}(C_{bi})$ contains at most $2\ell-1$ responses. Recall that $\mathcal{D}$ is a strengthened (m, ℓ)-BD object that outputs at most two distinct values as long as it is not upset. Thus $\mathcal{D}.M_{RES}(C_{bi})$ contains at most two distinct values. So the mode v_{mode} of $\mathcal{D}.M_{RES}(C_{bi})$ appears at least $\lceil |\mathcal{D}.M_{RES}(C_{bi})|/2 \rceil$ times. Thus, the number of responses in $\mathcal{D}.M_{RES}(C_{bi})$ that are *not* v_{mode} is at most $\lfloor |\mathcal{D}.M_{RES}(C_{bi})|/2 \rfloor \leq \lfloor (2\ell - 1)/2 \rfloor = \ell - 1$. ◀

▶ **Claim 30.** *There exists a pair of distinct processes p_0^* and p_1^*, and a value v_1^* (not necessarily distinct from v_{mode}), such that starting from C_{bi},*

- *p_0^* can take a step with response v_{mode}, and the resulting configuration C_0^* is univalent.*
- *p_1^* can take a step with response v_1^*, and the resulting configuration C_1^* is univalent, with different valence from C_0^*.*

Proof. Consider the processes p_0 and p_1, the values v_0 and v_1, and the configurations C_0 and C_1 stated in Claim 26. If $v_0 = v_{mode}$ or $v_1 = v_{mode}$, we are done.

Suppose $v_0 \neq v_{mode}$ and $v_1 \neq v_{mode}$. Let C'_0 be the configuration reached when, starting from C_{bi}, p_0 takes a step with response v_{mode} (note that this step is possible, since the number of distinct response values, and the number of responses different from v_{mode}, do not increase). Similarly, let C'_1 be the configuration reached when, starting from C_{bi}, p_1 takes a step with response v_{mode}. If C'_0 and C'_1 have different valence, we are done.

Suppose C'_0 and C'_1 have the same valence. Without loss of generality, suppose they are both 0-valent. Observe that, starting from C_{bi}, the 0-valent configuration C'_0 is reached when p_0 takes a step with response v_{mode}, and the 1-valent configuration C_1 is reached when p_1 takes a step with response v_1. ◀

We claim that:

- Starting from C_0^*, p_1^* can take a step with response v_1^*, and the resulting configuration $\widehat{C}_0^*$ is univalent.
- Starting from C_1^*, p_0^* can take a step with response v_{mode}, and the resulting configuration $\widehat{C}_1^*$ is univalent, with different valence from $\widehat{C}_0^*$.

To see why, note that in both $\widehat{C}_0^*$ and $\widehat{C}_1^*$, the number of distinct response values from $\mathcal{D}$ is at most two, and from Claim 29, the number of responses different from v_{mode} is at most $(\ell - 1) + 1 = \ell$. Thus the above steps required to reach $\widehat{C}_0^*$ and $\widehat{C}_1^*$ are possible.

Recall that the state of $\mathcal{D}$ does not record information about the order of operations and responses: it records only the set of proposals and the multiset of responses so far. Consequently, the state of $\mathcal{D}$ is the same in $\widehat{C}_0^*$ as in $\widehat{C}_1^*$. Furthermore, since p_0^* received v_{mode} as the response from $\mathcal{D}$ in both $\widehat{C}_0^*$ and $\widehat{C}_1^*$, the state of p_0^* is the same in $\widehat{C}_0^*$ as in $\widehat{C}_1^*$. Similarly, the state of p_1^* is the same in $\widehat{C}_0^*$ as in $\widehat{C}_1^*$. Finally, observe that all other objects and processes have not changed their state since C_{bi}. Thus their states are also the same in $\widehat{C}_0^*$ as in $\widehat{C}_1^*$. Therefore $\widehat{C}_0^* = \widehat{C}_1^*$, contradicting the fact that $\widehat{C}_0^*$ and $\widehat{C}_1^*$ have opposite valence. ◀

We can now determine the consensus number for every BD task and object as follows:

▶ **Theorem 31.** *For all $\ell \geq 0$ and $m > 0$, both the (m, ℓ)-BD task and the (m, ℓ)-BD object have consensus number $\max(m - 2\ell, 1)$.*

Proof. Consider any $\ell \geq 0$ and $m > 0$. There are two cases: either $m > 2\ell$, or $0 \leq m \leq 2\ell$.

First, consider the case where $m > 2\ell$. Since BD objects are at least as strong as their corresponding BD tasks (Observation 19), by Corollary 15 both the (m, ℓ)-BD task and the (m, ℓ)-BD object can, together with registers, solve the $(m - 2\ell)$-consensus task. Similarly, by Theorem 20 both the (m, ℓ)-BD task and the (m, ℓ)-BD object, together with registers, cannot solve the $(m - 2\ell + 1)$-consensus task. Thus both the (m, ℓ)-BD task and the (m, ℓ)-BD object have consensus number $m - 2\ell = \max(m - 2\ell, 1)$.

Now consider the case where $m \leq 2\ell$. From the above case, we have that both the $(2\ell + 1, \ell)$-BD task and the $(2\ell + 1, \ell)$-BD object have consensus number 1. For $m \leq 2\ell$, it is clear that the $(2\ell+1, \ell)$-BD object is at least as strong as the (m, ℓ)-BD object, and the $(2\ell+1, \ell)$-BD task is at least as difficult to solve as the (m, ℓ)-BD task. Thus both the (m, ℓ)-BD task and the (m, ℓ)-BD object also have consensus number $1 = \max(m - 2\ell, 1)$. ◀

5 An Unusual Property of Bounded Disagreement

We now show that every level $n \geq 2$ of Herlihy's consensus hierarchy contains a BD object that cannot be implemented using n-consensus and registers.

▶ **Theorem 32.** *For all $n \geq 2$, both the $(9n, 4n)$-BD task and the $(9n, 4n)$-BD object have consensus number n, but cannot be implemented using n-consensus objects and registers.*

Proof. Let $n \geq 2$, and consider the (m, ℓ)-BD object for $m = 9n$ and $\ell = 4n$.

1. Since $\ell \geq 0$ and $m > 0$, by Theorem 31, the $(9n, 4n)$-BD object has consensus number $\max(m - 2\ell, 1) = 9n - 2(4n) = n$.
2. Consider the (n', k)-SA task for $n' = 3n$ and $k = 2$. Note that $m = n' + \ell + \lfloor \ell/k \rfloor$ (because $9n = 3n + 4n + \lfloor 4n/2 \rfloor$), and $m \geq 2\ell + 2$ (because for $n \geq 2$, $9n \geq 2(4n) + 2$). Then, since $k \geq 1$, $n' \geq 1$, $\ell \geq 0$, and $m \geq \max(n' + \ell + \lfloor \ell/k \rfloor, 2\ell + 2)$, by Theorem 14 and the fact that the $(9n, 4n)$-BD object is at least as strong as the $(9n, 4n)$-BD task (Observation 19), $(9n, 4n)$-BD objects and registers can solve the $(3n, 2)$-SA task.

We claim that the $(9n, 4n)$-BD object cannot be implemented using n-consensus objects and registers. Suppose, for contradiction, that n-consensus objects and registers can implement the $(9n, 4n)$-BD object. Since $(9n, 4n)$-BD objects and registers can solve the $(3n, 2)$-SA task, this implies that n-consensus objects and registers can solve the $(3n, 2)$-SA task. This contradicts the following important result about set agreement: for all $n \geq 2$, n-consensus objects and registers cannot solve the $(3n, 2)$-SA task [8].

Thus, the $(9n, 4n)$-BD object has consensus number n, but cannot be implemented using n-consensus objects and registers. The proof that this also holds for the $(9n, 4n)$-BD *task* is almost identical. ◀

Prior to this paper, the only known objects with consensus number $n \geq 2$ and not implementable using n-consensus and registers were the ad hoc objects defined by Rachman [18] and Afek *et al.* [2]. In the appendix, we prove that these objects are fundamentally different from our BD objects.

6 Conclusion

In this paper, we introduced a novel and natural generalization of consensus that we call bounded disagreement, and investigated its relation to consensus and set agreement. Our results show that, despite apparent similarities, bounded disagreement is fundamentally different from these two problems:

- *BD and consensus are distinct problems.* In fact, by Theorem 32, for all $n \geq 2$, the $(9n, 4n)$-BD task [object] have consensus number n, but cannot be solved [implemented] by n-consensus and registers. Thus for all $k \leq n$, k-consensus and registers cannot solve the $(9n, 4n)$-BD task, and for all $k > n$, the $(9n, 4n)$-BD task and registers cannot solve k-consensus: so, for all $n \geq 2$, there is no k such that k-consensus is equivalent to the $(9n, 4n)$-BD task. Therefore there are infinitely many instances of the bounded disagreement task that are not equivalent to any consensus task.
- *BD and SA are distinct problems.* Except for the special case of $k = 1$ where the (n, k)-SA task is simply the n-consensus task, the (n, k)-SA task and registers cannot solve the 2-consensus task [8]. In contrast, by Theorem 31, the $(4, 1)$-BD task has consensus number 2, and in fact, for all $n \geq 2$, there are $\ell \geq 1$ and m such that the (m, ℓ)-BD task has consensus number n. So there are infinitely many instances of the bounded disagreement task that are not equivalent to any set agreement task.

Consensus is one of the fundamental problems in distributed computing, and its generalizations tend to produce deep and valuable insights about the field as a whole. The history of set agreement is one such example: since its introduction in 1990 [7], research on this problem has led to significant results, such as linking topology and distributed computing [17, 19]. It has also elucidated the relationship between consensus and other important problems in distributed computing, such as renaming and symmetry breaking [6, 13, 14]. The bounded disagreement problem may provide a similar impetus in the future.

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, Sep 1993. doi:10.1145/153724.153741.

2 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC'16, pages 97–106, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933116.

3 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007. doi:10.1007/s00446-007-0023-3.

4 Yehuda Afek, Adam Morrison, and Guy Wertheim. From bounded to unbounded concurrency objects and back. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'11, pages 119–128, New York, NY, USA, 2011. ACM. doi:10.1145/1993806.1993823.

5 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC'93, pages 159–170, New York, NY, USA, 1993. ACM. doi:10.1145/164051.164071.

6 Armando Castañeda, Damien Imbs, Sergio Rajsbaum, and Michel Raynal. Renaming is weaker than set agreement but for perfect renaming: A map of sub-consensus tasks. In *Proceedings of the 10th Latin American International Conference on Theoretical Informatics*, LATIN'12, pages 145–156, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-29344-3_13.

7 Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC'90, pages 311–324, New York, NY, USA, 1990. ACM. doi:10.1145/93385.93431.

8 Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the borowsky-gafni simulation (extended abstract). In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG'96, pages 362–379, London, UK, UK, 1996. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645953.675630.

9 Matei David. A single-enqueuer wait-free queue implementation. In Rachid Guerraoui, editor, *Distributed Computing*, volume 3274 of *Lecture Notes in Computer Science*, pages 132–143. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30186-8_10.

10 Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 137–151, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11561927_12.

11 David Eisenstat. Two-enqueuer queue in common2. *CoRR*, abs/0805.0444, 2008. URL: http://arxiv.org/abs/0805.0444.

12 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr 1985. doi:10.1145/3149.214121.

13 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 329–338. Springer Berlin Heidelberg, 2006. doi:10.1007/11864219_23.

14 Eli Gafni, Michel Raynal, and Corentin Travers. Test&set, adaptive renaming and set agreement: a guided visit to asynchronous computability. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 93–102, Oct 2007. doi:10.1109/SRDS.2007.8.

15 Maurice Herlihy. Impossibility results for asynchronous pram (extended abstract). In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'91, pages 327–336, New York, NY, USA, 1991. ACM. doi:10.1145/113379.113409.

16 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 11(1):124–149, Jan 1991. doi:10.1145/114005.102808.

17 Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects (preliminary version). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'94, pages 324–333, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198119.

18 Ophir Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG'94, pages 156–163, London, UK, UK, 1994. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645951.675479.

19 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, Mar 2000. doi:10.1137/S0097539796307698.

A BD Objects Versus the Object Defined by Rachman in [18]

In [18], Rachman defined the 2-*set object with parameter* m, which is a composite of an $(\infty, 2)$-SA object and an m-consensus object. Rachman proved that this object has consensus number m but cannot be implemented by m-consensus objects and registers.

▶ **Theorem 33.** *For all $m \geq 1$, the 2-set object with parameter m cannot be implemented by (n, ℓ)-BD objects and registers for any $n \geq 1$ and $\ell \geq 0$.*

Proof. For all $n \geq 1$, n-consensus objects and registers cannot solve the $(\infty, 2)$-SA task [8]. Since the n-consensus object can trivially implement the (n, ℓ)-BD object for all $\ell \geq 0$, (n, ℓ)-BD objects and registers cannot solve the $(\infty, 2)$-SA task. In contrast, by definition, the 2-set object with parameter $m \geq 1$ solves the $(\infty, 2)$-SA task [18]. ◀

B BD Objects Versus the Object Defined by Afek *et al.* in [2]

In [2], Afek *et al.* defined a family of deterministic objects $O_{m,k}$, where $m \geq 2$ and $k \geq 2$, such that: (a) $O_{m,k}$ has consensus number m, (b) $O_{m,k}$ solves the $(km + k - 1, k)$-SA task, and (c) $O_{m,k+1}$ implements $O_{m,k}$. They also proved that $O_{m,k}$ objects and registers cannot solve the $(km + m + k, k + 1)$-SA task, which is trivially solved by the $O_{m,k+1}$ object since $km + m + k = (k + 1)m + (k + 1) - 1$.

▶ **Theorem 34.** *For all $n \geq 1$, the $(9n, 4n)$-BD object is not equivalent to the $O_{m,k}$ object, for any $m \geq 2$ and $k \geq 2$.*

Proof. Consider any $n \geq 1$, $m \geq 2$, and $k \geq 2$. By Theorem 31, the $(9n, 4n)$-BD object has consensus number n. On the other hand, the $O_{m,k}$ object has consensus number m [2]. If $m \neq n$, it is clear that the two objects are not equivalent. So suppose that $n = m$.

Case 1: k is odd. Thus there exists a positive integer q such that $k + 1 = 2q$. Observe that:

- Any solution to the $(2q(m+1), 2q)$-SA task can be used to solve the $(km+m+k, k+1)$-SA task; this is because $km + m + k = (k + 1)(m + 1) - 1 = 2q(m + 1) - 1 < 2q(m + 1)$.
- Any solution to the $(2(m+1), 2)$-SA task can be used, together with registers, to solve the $(2q(m + 1), 2q)$-SA task; the algorithm to do so is obvious: divide the $2q(m + 1)$ processes into q groups of $2(m + 1)$ processes, and have each group solve the $(2(m + 1), 2)$-SA task independently [8].

- Since $m \geq 2$, $2(m+1) \leq 3m$, and so any solution to the $(3m, 2)$-SA task can be used to solve the $(2(m+1), 2)$-SA task.
- Since $m = n$, the $(3m, 2)$-SA task is the $(3n, 2)$-SA task.
- By Theorem 14 and the fact that the $(9n, 4n)$-BD object is at least as strong as the $(9n, 4n)$-BD task (Observation 19), $(9n, 4n)$-BD objects and registers can solve the $(3n, 2)$-SA task.

Thus, by transitivity, we have that $(9n, 4n)$-BD objects and registers can solve the $(km + m + k, k + 1)$-SA task. In contrast, as we mentioned above, $O_{m,k}$ objects and registers cannot solve the $(km + m + k, k + 1)$-SA task [2]. We conclude that the $(9n, 4n)$-BD object cannot be implemented using $O_{m,k}$ objects and registers.

Case 2: k is even. Then $k + 1$ is odd. From case 1, the $(9n, 4n)$-BD object cannot be implemented using $O_{m,k+1}$ objects and registers. Since the $O_{m,k+1}$ object implements the $O_{m,k}$ object [2], by transitivity, the $(9n, 4n)$-BD object cannot be implemented using $O_{m,k}$ objects and registers. ◀

Read-Write Memory and k-Set Consensus as an Affine Task

Eli Gafni[1], Yuan He[2], Petr Kuznetsov*[3], and Thibault Rieutord*[4]

1 **UCLA, Los Angeles, CA, USA**
eli@ucla.edu
2 **UCLA, Los Angeles, CA, USA**
yuan.he@ucla.edu
3 **Télécom ParisTech, Paris, France**
petr.kuznetsov@telecom-paristech.fr
4 **Télécom ParisTech, Paris, France**
thibault.rieutord@telecom-paristech.fr

Abstract

The wait-free read-write memory model has been characterized as an iterated *Immediate Snapshot* (IS) task. The IS task is *affine* – it can be defined as a (sub)set of simplices of the standard chromatic subdivision. In this paper, we highlight the phenomenon of a "natural" model that can be captured by an iterated affine task and, thus, by a subset of runs of the iterated immediate snapshot model. We show that the read-write memory model in which, additionally, k-set-consensus objects can be used is "natural" by presenting the corresponding simple affine task captured by a subset of 2-round IS runs. As an "unnatural" example, the model using the abstraction of *Weak Symmetry Breaking* (WSB) cannot be captured by a set of IS runs and, thus, cannot be represented as an affine task. Our results imply the first combinatorial characterization of models equipped with abstractions other than read-write memory that applies to generic tasks.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation

Keywords and phrases iterated affine tasks, k-set consensus, k-concurrency, simplicial complexes, immediate snapshot

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.6

1 Introduction

A principal challenge in distributed computing is to devise protocols that operate correctly in the presence of failures, given that system components (processes) are asynchronous.

The most extensively studied *wait-free* model of computation [21] makes no assumptions about the number of failures that can occur. In particular, in a *wait-free* solution of a distributed *task*, a process participating in the computation should be able to produce an output regardless of the behavior of other processes.

Topology of wait-freedom. Wait-free task solvability in the read-write shared-memory model has been characterized in an elegant way through the existence of a specific continuous map from geometrical structures describing inputs and outputs of the task [22, 24].

* This work has been partially supported by the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.

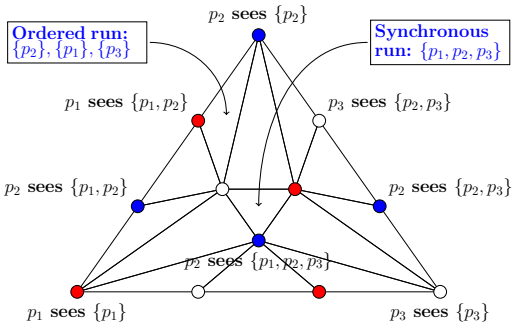


Figure 1 Chr $\mathbf{s}$ in the 2-dimensional case.

A task T is *wait-free solvable* (using reads and writes) if and only if there exists a simplicial, chromatic map from a subdivision of the *input* simplicial complex to the *output* simplicial complex, satisfying the specification of T. In particular, using the iterated *standard chromatic subdivision* [22, 26] (one such iteration of the *standard* 2-*simplex* $\mathbf{s}$, denoted by Chr $\mathbf{s}$, is depicted in Figure 1), we obtain a combinatorial representation of the wait-free model. *Iterations* of this subdivision capture precisely rounds of the iterated immediate snapshot (IIS) model [5, 24].

This characterization can be interpreted as follows: the wait-free read-write model can be characterized, regarding task solvability, by an *iterated* (one-shot) Immediate Snapshot task, which, in turn, captured by the *chromatic simplex agreement* task [5, 24] on Chr $\mathbf{s}$.

Beyond wait-freedom: k-concurrency and k-set consensus. Unfortunately, very few tasks are solvable in the wait-free manner using read and writes [3, 24, 31], so a lot of work has been invested in characterizing task solvability in various *restrictions* of the *wait-free* model.

A straightforward way to define such a restriction is to bound the *concurrency level* of runs [14]. In the k-*concurrency* model, at most k processes can be concurrently *active* (during the interval between the invocation and response of the task). The k-concurrency model is known to be equivalent to the k-*set-consensus* model in which processes, in addition to the read-write shared memory, can access k-set-consensus objects [13].

Iterated tasks for k-set-consensus. In this paper, we show that the k-set-consensus model can be captured by an iterated *affine* task [17]. Informally, an affine task consists in solving chromatic simplex agreement [5, 24] on a *subcomplex* of some iteration of the standard chromatic subdivision. We show that the affine task $\mathcal{R}_k$ capturing the k-set-consensus model consists of all simplices of the second chromatic subdivision, in which at most k processes *contend* with each other (cf. examples of $\mathcal{R}_1$ and $\mathcal{R}_2$ for 3 processes in Figure 3).

We show that $\mathcal{R}_k^*$, the set of IIS runs corresponding to iterations of this subcomplex $\mathcal{R}_k$, solves precisely the same set of tasks as the k-set-consensus model does. Note that our definition of the IIS model does not assume process failures: every process takes infinitely many steps in every run, but, because of the use of iterated memory, a "slow" process may not be seen by "faster" ones from some point [29, 10, 7]. Thus, solving a task in $\mathcal{R}_k^*$ requires *every* process to output.

Techniques and results. Our result is established through the existence of three *simulations*.

The first one simulates, in the k-set-consensus model, a k-concurrent execution of any read-write algorithm in the k-set-consensus model. We derive that the k-set-consensus model solves every task that is solvable k-concurrently.

The second algorithm *solves* $\mathcal{R}_k$ in the k-concurrency model, i.e., solves chromatic simplex agreement on the $\mathcal{R}_k$ complex. By iterating this solution, we can *simulate* the $\mathcal{R}_k^*$ model and, thus, solve any task solvable in $\mathcal{R}_k^*$.

The third algorithm simulates runs of an algorithm using read-write memory and k-set-consensus objects in $\mathcal{R}_k^*$. Compared to the simulations in [23, 18, 16, 7, 17], our algorithm ensures that every process eventually outputs in $\mathcal{R}_k^*$, assuming that the simulated algorithm ensures that every *correct* process eventually outputs.

Thus, a task is solvable using iterations of $\mathcal{R}_k$ *if and only if* it can be solved in the wait-free model using reads, writes, and k-set-consensus objects (or, equivalently, assuming k-concurrency). Therefore, the k-set-consensus model has a bounded representation as an *iterated affine task*: processes iteratively invoke instances of $\mathcal{R}_k$ a bounded number of times until they assemble enough knowledge to produce an output for the task they are solving.

Our results suggest a separation between "natural" models that have matching affine tasks and, thus, can be captured precisely by subsets of IIS runs and less "natural" ones, like WSB, having a manifold structure that is not affine [19]. We conjecture that such a combinatorial representation can also be found for a large class of restrictions of the wait-freedom, beyond k-concurrency and k-set consensus. This claim is supported by a recent derivation of the t-resilience affine task [32].

Related work. There have been several attempts to extend the topological characterization of [24] to models beyond the wait-free one [23, 16, 17, 28]. However, these results either only concern the special case of *colorless* tasks [23], consider weaker forms of solvability [16], introduce a new kind of *infinite* subdivisions [17], or also use non-iterated infinite subset of IIS runs [28].

In particular, Gafni et al. [17] characterized task solvability in models represented as subsets of IIS runs via *infinite* subdivisions of input complexes. This result assumes a limited notion of task solvability in the iterated model that only guarantees outputs to "fast" processes [11, 29, 7] that are "seen" by every other process infinitely often.

Concerning the reduction to iterated models, Imbs et al [25] showed that the model of iterated x-consensus, i.e., consensus among x processes, is equivalent regarding task solvability to the model of read-write memory and access to x-consensus objects.

In contrast with the earlier work, this paper studies the inherent combinatorial properties of general (colored) tasks and assumes the conventional notion of task solvability. Concurrently with the recently discovered affine task for *t-resilience* [32], our results truly capture the combinatorial structure of a restriction of the wait-free model.

Roadmap. The rest of the paper is organized as follows. Section 2 gives model definitions, briefly overviews the topological representation of iterated shared-memory models. In Section 3, we present the definition of $\mathcal{R}_k$ corresponding to the k-concurrency model. In Section 4, we show that $\mathcal{R}_k$ can be implemented in the k-set-consensus model and that any task solvable in the k-set-consensus model can be solved by iterating $\mathcal{R}_k$. Section 5 discusses related models and open questions.

2 Preliminaries

Let Π be a system composed of n asynchronous processes, $p_1, \ldots, p_n$. We consider two models of communication: (1) *atomic snapshots* [1] and (2) *iterated immediate snapshots* [5, 24].

Atomic snapshots. The atomic-snapshot (AS) memory is represented as a vector of shared variables, where processes are associated to distinct vector positions, and exports two operations: *update* and *snapshot*. An *update* operation performed by p_i replaces the shared variable at position i with a new value and a *snapshot* returns the current state of the vector. The model in which processes only have access to an AS memory is called the AS model.

Iterated immediate snapshots. In the iterated immediate snapshot (IIS) model, processes proceed through an ordered sequence of independent memories $M_1, M_2, \ldots$. Each memory M_r is accessed by a process with a single *immediate snapshot* operation [4]: the operation performed by p_i takes a value v_i and returns a set V_{ir} of values submitted by the processes (w.l.o.g, we assume that values submitted by different processes are distinct), so that the following properties are satisfied: (self-inclusion) $v_i \in V_{ir}$; (containment) $(V_{ir} \subseteq V_{jr}) \vee (V_{jr} \subseteq V_{ir})$; and (immediacy) $v_i \in V_{jr} \Rightarrow V_{ir} \subseteq V_{jr}$.

Protocols and runs. A *protocol* is a distributed automaton that, for each local state of a process, stipulates which operation and which state transition the process is allowed to perform in its next step. We assume here *deterministic* protocols, where only one operation and state transition is allowed in each state. A *run* of a protocol is defined as a possibly infinite sequence of states and operations.

In the IIS communication model, we assume that processes run the *full-information* protocol: the first value each process writes is its *initial state*. For each $r > 1$, the outcome of the immediate snapshot operation on memory M_{r-1} is submitted as the input value for the immediate snapshot operation on memory M_r. After a certain number of such (asynchronous) rounds, a process may gather enough information to *decide*, i.e., to produce an irrevocable non-$\perp$ output value. A *run* of the IIS communication model is thus a sequence V_{ir}, $i \in \mathbb{N}_n$ and $r \in \mathbb{N}$, determining the outcome of the immediate-snapshot operation for every process i and each iterated memory M_r.

Failures and participation. In the AS model (or the defined below AS model in which k-set-consensus objects can additionally be accessed), a process that takes only finitely many steps of the assigned protocol in a given run is called *faulty*, otherwise it is called *correct*. We assume that in its first step, a process writes its initial state in the shared memory using the *update* operation. If a process completed this first step it is said to be *participating*, the set of participating processes in a given run is called the *participating set*. Note that, since every process writes its initial state in its first step, the initial states of participating processes are eventually known to every process that takes sufficiently many steps.

In contrast, the IIS model does not have the notion of a faulty process. Instead, a process may appear "slow" [29, 10, 7], i.e., be late in accessing iterated memories from some point on so that some "faster" processes do not see them.

Tasks. In this paper, we focus on distributed *tasks* [24]. A process invokes a task with an *input* value and the task returns an *output* value, so that the inputs and the outputs across the processes, respect the task specification. Formally, a *task* is defined through a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. An input $\perp$ denotes a *non-participating* process and an output value $\perp$ denotes an *undecided* process. Check [22] for more details on the definition.

A protocol solves a task $T = (\mathcal{I}, \mathcal{O}, \Delta)$ in a given model, if it ensures that in every infinite run of the model in which processes start with an input vector $I \in \mathcal{I}$, there is a finite prefix R of the run where: (1) all decided values form a vector $O \in \mathcal{O}$ such that $(I, O) \in \Delta$, and (2) every correct process has decided.

k-set-consensus and k-concurrency models. A k-set-consensus object can be accessed with a *propose* operation that takes a *proposed* value as an argument and returns a *decided* value as a response, so that, in each run, decided values constitute a subset of proposed values of size at most k. For $k \in \{1, \ldots, n-1\}$, in the *k-set-consensus model*, processes communicate via the AS memory and k-ste-consensus objects.

Let R be a finite run of the AS model. A process p_i is called *active at the end of* R if p_i participates in R but has not terminated by the end of R. Let *active*(R) denote the set of all processes that are active at the end of R.

A run R of the AS model is *k-concurrent* ($k = 1, \ldots, n$) if at most k processes are *concurrently active* in R, i.e., $\max\{|\mathit{active}(R')|;\ R' \text{ prefix of } R\} \leq k$. The *$k$-concurrency* model is the set of k-concurrent runs of the AS model.

It is known that the k-concurrency model is *equivalent* to the k-set-consensus model [13][1]: any task that can be solved k-concurrently can also be solved in the k-set-consensus model, and vice versa.

Standard chromatic subdivision and IIS. To give a combinatorial representation of the IIS model, we use the language of *simplicial complexes* [33, 22]. In short, a simplicial complex is defined as a set of *vertices* and an inclusion-closed set of vertex subsets, called *simplices*. The dimension of a simplex σ is the number of its vertices minus one. Any subset of these vertices is called a *face* of the simplex. A simplicial complex is *pure* (of dimension n) if each its simplices are contained in a simplex of dimension n.

A simplicial complex is *chromatic* if it is equipped with a *coloring function* – a non-collapsing simplicial map χ from its vertices to the *standard* $(n-1)$*-simplex* $\mathbf{s}$ of n vertices, in one-to-one correspondence with n *colors* $1, 2, \ldots, n$. All simplicial complexes we consider here are pure and chromatic. Please refer to the extended version of this paper [15] or [22] for more details on the formalism.

For a chromatic complex C, we let $\mathrm{Chr}\, C$ be the subdivision of C obtained by replacing each simplex in C with its *standard chromatic subdivision* [26]. The vertices of $\mathrm{Chr}\, C$ are pairs (v, σ), where p is a vertex of C and σ is a simplex of C containing v. Vertices (v_1, σ_1), …, (v_m, σ_m) form a simplex if all v_i are distinct and all σ_i satisfy the properties of immediate snapshots. Subdivision $\mathrm{Chr}^1\, \mathbf{s}$ for the 2-dimensional simplex $\mathbf{s}$ is given in Figure 1. Each vertex represents a local state of one of the three processes p_1, p_2 and p_3 (red for p_1, blue for p_2 and white for p_3) after it takes a single immediate snapshot. Each triangle (2-simplex) represents a possible state of the system. A corner vertex corresponds to a local state in which the corresponding process only sees itself (it took its snapshot before the other two processes moved). An interior vertex corresponds to a state in which the process sees all three processes. The vertices on the 1-dimensional faces capture the snapshots of size 2.

If we *iterate* this subdivision m times, each time applying the same subdivision to each of the simplices, we obtain the m^{th} chromatic subdivision, $\mathrm{Chr}^m\, C$. It turns out that $\mathrm{Chr}^m\, \mathbf{s}$ precisely captures the m-round (full-information) IIS model, denoted IS^m [24]. Each run of IS^m corresponds to a simplex in $\mathrm{Chr}^m\, \mathbf{s}$. Every vertex v of $\mathrm{Chr}^m\, \mathbf{s}$ is thus defined

[1] In fact, this paper contains a self-contained proof of this equivalence result.

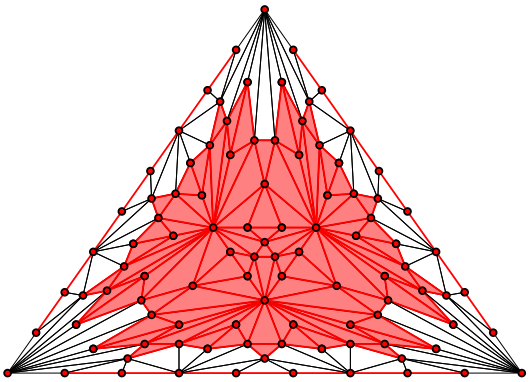


Figure 2 Contention sets (simplices in red) in a 3-process system.

as $(p, IS^1(p,\sigma), \ldots, IS^m(p,\sigma))$, where each $IS^i(p,\sigma)$ is interpreted as the set of processes appearing in the i^{th} IS iteration obtained by p in the corresponding IS^m run. The *carrier* of vertex v is then defined as the set of all processes seen by p in this run, possibly through the views of other processes: it is the smallest face of $\mathbf{s}$ that contains v in its geometric realization. The carrier of a simplex is the maximal carrier of its vertices (related by inclusion).

Affine Tasks. As we show in this paper, the k-set-consensus model (and, thus, the k-concurrency model) can be captured by an iterated *affine* task [17]. Affine tasks can be seen as a generalization of simplex agreement tasks [5, 24], where the output complex is no longer a subdivision but a subset of some iteration of the standard chromatic subdivision. More formally, let L be a pure subcomplex of $\mathrm{Chr}^l \mathbf{s}$ for some $l \in \mathbb{N}$ of the dimension of $\mathbf{s}$. The affine task associated to L is then simply defined as $(\mathbf{s}, L, \Delta)$, where, for every face $\mathbf{t} \subseteq \mathbf{s}$, $\Delta(\mathbf{t}) = L \cap \mathrm{Chr}^l \mathbf{t}$. With a slight abuse of notations, the subcomplex L is used to denote the affine task associated to L.

By running m iterations of this task, we obtain L^m, a subcomplex of $\mathrm{Chr}^{lm} \mathbf{s}$, corresponding to a subset of IS^{lm} runs (each iteration includes l IS rounds). We denote by L^* the set of infinite runs of the IIS model where every prefix restricted to a multiple m of l IS rounds belongs to the subset of IS^{lm} runs associated to L^m.

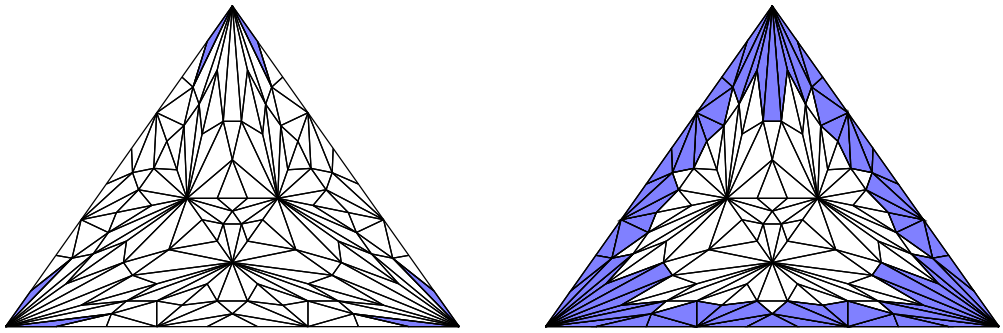
3 The complex of k-set consensus

We now define $\mathcal{R}_k$, a subcomplex of $\mathrm{Chr}^2 \mathbf{s}$, that precisely captures the ability of k-set consensus (and read-write memory) to solve tasks. The definition of $\mathcal{R}_k$ is expressed via a restriction on the simplices of $\mathrm{Chr}^2 \mathbf{s}$ that bounds the size of *contention sets*. Informally, a contention set of a simplex $\sigma \in \mathrm{Chr}^2 \mathbf{s}$ (or, equivalently, of an IS^2 run) is a set of vertices (or, processes) that "see each other". When a process p_i starts its IS^2 execution after another process p_j terminates, p_i must observe p_j's input, but not vice versa. Thus, a set of processes that see each others' inputs must have been concurrently active at some point.

Topologically speaking, a contention set of a simplex $\sigma \in \mathrm{Chr}^2 \mathbf{s}$ is a set of vertices in σ sharing the same carrier, i.e., a minimal face $\mathbf{t} \subseteq \mathbf{s}$ that contains their vertices. Thus, for a given simplex $\sigma \in \mathrm{Chr}^2 \mathbf{s}$, the set of contention sets is defined as follows:

▶ **Definition 1** (Contention sets). $Cont(\sigma) = \{\tau \subseteq \sigma, \forall v \in \tau, carrier(v) = carrier(\tau)\}$.

Contention sets for simplices of $\mathrm{Chr}^2 \mathbf{s}$ in a 3-process system are depicted in Figure 2: for each simplex $\sigma \in \mathrm{Chr}^2 \mathbf{s}$, every face of σ that constitutes a red simplex is a contention



(a) Complex $\mathcal{R}_1$ **(b)** Complex $\mathcal{R}_2$

Figure 3 $\mathcal{R}_1$ and $\mathcal{R}_2$ (in blue) for 3 processes.

set of σ. In an interior simplex, every set of vertices are contention sets. Every "total order" simplex (shown in blue in Figure 3a), matching a run in which processes proceed, one by one, in the same order in both IS^1 and IS^2, has only three singleton as contention sets. All other simplices include a contention set of two processes. In the associated IIS run, processes are said to be *contending* if they are associated to vertices forming a contention set.

Now $\mathcal{R}_k$ is defined as the set of all simplices in $\mathrm{Chr}^2\,\mathbf{s}$, in which the contention sets have cardinalities of at most k:

▶ **Definition 2** (Complex $\mathcal{R}_k$). $\mathcal{R}_k = \{\sigma \in \mathrm{Chr}^2\,\mathbf{s}, \forall \tau \in \mathit{Cont}(\sigma), |\tau| \leq k\}$.

It is immediate to see that the set of simplices in $\mathcal{R}_k$ constitutes a simplicial complex: every face τ of $\sigma \in \mathcal{R}_k$ is also in $\mathcal{R}_k$.

Examples of $\mathcal{R}_1$ and $\mathcal{R}_2$ for a 3-process system are shown in Figures 3a and 3b, respectively. Obviously, for the unrestricted 3-set consensus case, $\mathcal{R}_3 = \mathrm{Chr}^2\,\mathbf{s}$. Note that $\mathcal{R}_1$ only contains six "total order" simplices, while $\mathcal{R}_2$ consists of all simplices of $\mathrm{Chr}^2\,\mathbf{s}$ that touch the boundary.

4 From k-set consensus to $\mathcal{R}_k^*$ and back

In this section, we show that any task solvable in the k-set consensus model can be solved in $\mathcal{R}_k^*$, and vice versa. The main result is established via *simulations*: a run of an algorithm solving a task in one model is simulated in the other.

4.1 From k-set consensus to k-concurrency

We first show that the k-concurrency model is equivalent, regarding task solvability, to the k-set-consensus model. This result has been stated in a technical report [13], but no explicit proof appears in the literature, and we fill the gap below.

Simulating a k-process shared memory system. We employ *generalized state machines* (proposed in [14] and extended in [30]) that allow us to simulate an *inputless* (i.e., simulated processes have a default initial state) k-process read-write memory system in the k-set-consensus model. To ensure consistency of simulated operations, we use *commit-adopt* objects [11] that can be implemented using AS. A commit-adopt object exports one operation *propose*(v) that takes a parameter in an arbitrary range and returns a couple (flag, v'), where *flag* can be either *commit* or *adopt* and where v' is a previously proposed value. Moreover, if

Algorithm 1: k processes shared memory system simulation: process p_i.

```
1  SharedObjects: KSC[1...] k−simultaneous consensus objects;
2  CA[1...][1...k] : commit − adopt objects;
3  MEM[1...n][1...k] init (−1, ⊥) : single writer shared memory array;
4  Init: r_i ← 0; foreach m ∈ {1,...,k} do (WC_i[m], View_i[m]) ← (0, ∅);

5  Repeat forever
6      r_i ← r_i + 1;
7      (Index_i, Value_i) ← KSC[r_i].propose(WC, View_i);
8      (Flag_i[Index_i], val_i) ← CA[r_i][Index_i].propose(Value_i);
9      if val_i = (c, ∗) with c ≥ WC_i[Index_i] then (WC_i[Index_i], View_i[Index_i]) ← val_i;
10     foreach m ∈ {1,...,k} \ Index_i do
11         (Flag_i[m], val_i) ← CA[r_i][m].propose(View_i[m]);
12         if val_i = (c, ∗) with c ≥ WC_i[m] then (WC_i[m], View_i[m]) ← val_i;
13     foreach m ∈ {1,...,k} do
14         if Flag_i[m] = Commit then
15             MEM[i][m].Update(WC_i[m], WriteVal(WC_i[m], View_i[m]));
16             WC_i[m] ← WC_i[m] + 1;
17             View_i[m] = CurWrites(MEM.Snapshot());
18 End repeat;

19 With CurWrites (MEM_val)=
20     foreach m ∈ {1,...,k} do curWC[m] = −1, curWrite[m] = ⊥;
21     foreach (m, l) ∈ {1,...,k} × {1,...,n} do
22         if MEM_val[l][m].WC > curWC[m] then
23             curWC[m] = MEM_val[l][m].WC, curWrite[m] = MEM_val[l][m].Value;
24     return curWrite;
```

a process returns a *commit* flag, then every process must return the same value. Further, if no two processes propose different values, then all returned flags must be *commit*.

Liveness of the simulation relies on calls to *k-simultaneous consensus* objects [2]. To access a k-simultaneous consensus object, a process proposes a vector of k inputs, one for each of the *consensus instances*, $1, 2, \ldots, k$, and the object returns a couple(i, v), where *index* i belongs to $\{1, \ldots, k\}$ and v is a value proposed by some process at index i. It ensures that no two processes obtain different values with the same index. Moreover, if $\ell \leq k$ distinct input vectors are proposed then only values at indices $1, \ldots, \ell$ can be output. The k-simultaneous consensus object is equivalent to k-set-consensus in the read-write shared-memory system [2].

Our simulation is described in Algorithm 1. We use three shared abstractions: an infinite array of k-simultaneous consensus objects KSC, one object per round, an infinite array of arrays of k indexed commit-adopt objects CA, i.e., k objects per round, and a single-writer multi-reader memory MEM with k slots.

In every round, the simulator starts by accessing the round k-simultaneous-consensus object with its local estimation of the simulated system state (line 7). Then the simulator access the commit-adopt object associated with the index, and using the state estimation as proposed value, output earlier by the k-simultaneous-consensus object (line 8). After that the simulator go through the $k-1$ remaining commit-adopt objects of the round, using its current estimation of the simulated processes state as proposals (lines 8–11). It is guaranteed that at least one process commits, in particular, process p_j that is the first to return from its first commit-adopt invocation in this round (on a commit-object C), because any other process with a different proposal must access a different commit-adopt object first and, thus, it must invoke C after p_j returns. To ensure that a unique written value is selected, simulators replace their current proposal values with the values returned by the commit-adopt objects (lines 8–11). Note that the processes do not select values corresponding to an older round of simulation, to ensure that processes do not alternate committing and adopting the same value indefinitely.

In the simulation, the simulators propose snapshot results for the simulated processes. Once a proposed snapshot has been committed, a simulator stores in the shared memory the value that the simulated process must write in its next step (based on its simulated algorithm), equipped with the corresponding *write counter* (line 15). The write counter is then incremented and a new snapshot proposal is computed (line 17). To compute a simulated snapshot, for each process, the most recent value available in the memory MEM is selected by comparing the write counters WC (auxiliary function *CurWrites* at lines 19–24).

▶ **Lemma 3.** *Algorithm 1 provides a non-blocking simulation of an inputless k-process AS memory system in the k-set consensus model. Moreover, if there are $\ell < k$ active processes, then one of the first ℓ simulated processes is guaranteed to make progress.*

The proof of Lemma 3 can be found in the companion technical report [15]. The proof is constructed by showing that: (1) No two different written values are computed for the same simulated process and the same write counter; (2) In every round of the simulation, at least one simulator commits a new simulated operation; (3) Every committed simulated snapshot operation can be linearized at the moment when the actual snapshot operation which served for its computation took place; and (4) Every simulated write operation can be linearized to the linearization time of the first actual write performed by a simulator with the corresponding value.

Using the extended BG-simulation to simulate a k-concurrent execution. We have shown that a k-process inputless AS memory system can be simulated in the k-set-consensus model. In its turn, the simulated system can be used to simulate a k-concurrent execution by running an extended *BG-simulation* protocol [3, 6].

The BG-simulation technique allows $k+1$ processes $s_1, \ldots, s_{k+1}$, called *BG-simulators*, to simulate, in a wait-free manner, a *k-resilient* run of any protocol $\mathcal{A}$ on m processes $p_1, \ldots, p_m$ ($m > k$). The simulation guarantees that each simulated step of every process p_j is either agreed upon by all simulators, or is *blocked* because of a slow or faulty simulator (and one less simulator participates further in the simulation for each step which is not agreed on).

In the original BG simulation, a faulty simulator may indefinitely block an arbitrary simulated process. The *extended BG-simulation* [12] additionally exports an *abort* operation that, when applied to a blocked simulated process, re-initializes it, so that the process can move forward until an output for it is computed, or another simulator makes it block again. The abort mechanism must be used carefully in order to keep liveness properties, by, for example, ensuring that the slow or crashed simulator will not participate again in the simulation. Refer [3, 6, 12] for a more formal description of the BG-simulation technique.

As previously done in [9], the main idea of the simulation is to run a *depth-first* BG-simulation (i.e., simulate the more advanced available code) instead of a classical *bread-first* BG-simulation (i.e., simulate the least advanced available code). Indeed, the BG-simulation consists in executing pieces of shared-memory codes in any valid order (i.e., respecting simulated threads dependencies). But to prevent starvation of the BG-simulators, one cannot wait to simulate the next piece of code a blocked thread and thus must move to another thread. But by selecting the more advanced available thread, then, as we will verify, this provides a k-concurrent simulation when executed by k BG-simulators.

Three aspects requires clarification. First, the threads we want to execute k-concurrently require input values, initially available only to the corresponding process. This is resolved by simply having processes write their initial state to the shared memory before participating in the BG-simulation executed on the k-process inputless system provided by Algorithm 1.

Algorithm 2: Process code for the k-concurrent simulation for p_i.

```
1 Thread[i] ← Input;
2 while Thread[i]!=Complete do
3 |   Execute 1 round of Algorithm 1(Algorithm 3);
4 return Thread[i].output();
```

Algorithm 3: Code for BG-simulator number m.

```
1 Repeat forever
2 |   Let A = {i ∈ {1,...,n}, Thread[i]! = NotInitialized ∧ Thread[i]! = Complete};
3 |   if |A| > m then
4 |   |   Let Blocked = {i ∈ A, Thread[i] = Blocked};
5 |   |   if A = Blocked then
6 |   |   |   forall i ∈ A do Thread[i].Abort() ;
7 |   |   Let j = argmax_{i∈A\Blocked} Thread[i].Progress();
8 |   |   Execute(Thread[i].NextStep());
9 End repeat;
```

This way there is at least as many *opened* threads (i.e., a thread associated with a process with an input value visible to all simulators) as possible active BG-simulators.

Secondly, as threads are used to execute an algorithm solving a task, they may terminate, thus reducing the number of threads available. This is why processes execute Algorithm 1 one round at a time: At each round each process checks if its own thread is complete, and if so it stops and it returns with its task output. This ensures that the number of processes active in Algorithm 1 execution follows, with some latency, the number of threads available.

Lastly, when there is $\ell < k$ available threads, the last $(k - \ell)$ BG-simulators stop participating in the simulation to adapt the number of active BG-simulators to the number of available threads. Moreover, a BG-simulator uses the abort mechanism, on all active threads, if it is one of the first ℓ BG-simulators and if every active thread is currently blocked.

These algorithms are available in Algorithm 2 for the process code, and in Algorithm 3 for the BG-simulator code. The threads correspond to process simulations. A thread is said to be *NotInitialized* if the corresponding process did not provide yet its input value (Alg. 2, l. 1). Otherwise it can be either *Blocked*, *Available*, or *Complete*. If it is initialized, *Progress()* returns the number of simulation steps that has been performed, possibly 0. Note that in the case where multiple threads have the same progress when selecting the most advanced one (Alg. 3, l. 7), any one of them can be selected.

▶ **Lemma 4.** *All tasks solvable in the k-concurrency model can be solved in the k-set-consensus model.*

The proof of Lemma 4 is quite tedious, and therefore relegated to the associated technical report [15], even if the construction of the algorithms and the main aspect of their correctness are quite simple and natural.

The main argument relies on the fact that the number of processes participating in Algorithm 1 follows the number of active threads. Thus, when the number of active threads ℓ is smaller than k, we have the property that Algorithm 1 provides progress to one out of the first ℓ simulated processes (see Lemma 3). Therefore, one out of the ℓ first BG-simulators takes infinitely many steps. But as only the first ℓ BG-simulators remains active, the ℓ active threads cannot remain blocked by slow BG-simulators. The technical difficulties relies in showing that the use of the abort mechanism does not cause trouble as well as the delays to reach stability after a change in the number of active threads. On another hand, the safety

of the simulation is quite easy to show, as at most $k-1$ active threads may be blocked when selecting a thread to simulate, if k already took steps and did not terminate then one of these k will be selected to continue further its simulation.

4.2 From k-concurrency to $\mathcal{R}_k$.

We now show that k-concurrency can *solve* $\mathcal{R}_k$, i.e., it can solve the affine task on the subcomplex $\mathcal{R}_k$. In fact, running any algorithm implementing immediate snapshot in the k-concurrency model would do the job.

▶ **Lemma 5.** *The two-round immediate snapshot algorithm solves the affine tasks $\mathcal{R}_k$ in the k-concurrency model.*

Proof. Consider two iterated rounds of any IS algorithm (e.g., [4]) in the k-concurrency model. The set of IS^2 outputs of this algorithm, forms a valid simplex σ in $\mathrm{Chr}^2\,\mathbf{s}$ [5].

Let σ be any such simplex. Let S be the contention set of σ containing two vertices v and v', and let p and q be the processes corresponding to the vertices v and v'. By contradiction, suppose that p and q were not concurrently active in the corresponding IS^2 run. Without loss of generality, suppose that p's computation was terminated before the activation of q, so p cannot be aware of q's input. Thus, p cannot see q, which implies that v and v' have different carriers, which contradicts the definition of a contention set.

Therefore, all vertices in a contention set are associated to processes that were *active* at the same time. Thus, k-concurrent runs produce simplices in which contention sets are of size at most k and, thus, belong to $\mathcal{R}_k$. Since the simplices of $\mathcal{R}_k$ output by the simulation correspond to the participating processes only, we indeed get an algorithm solving chromatic simplex agreement on complex $\mathcal{R}_k$. ◀

Now it is easy to show that the k-set consensus model is, regarding task solvability, at least as strong as the $\mathcal{R}_k^*$ model:

▶ **Theorem 6.** *Any task solvable by $\mathcal{R}_k^*$ can be solved in the k-set-consensus model.*

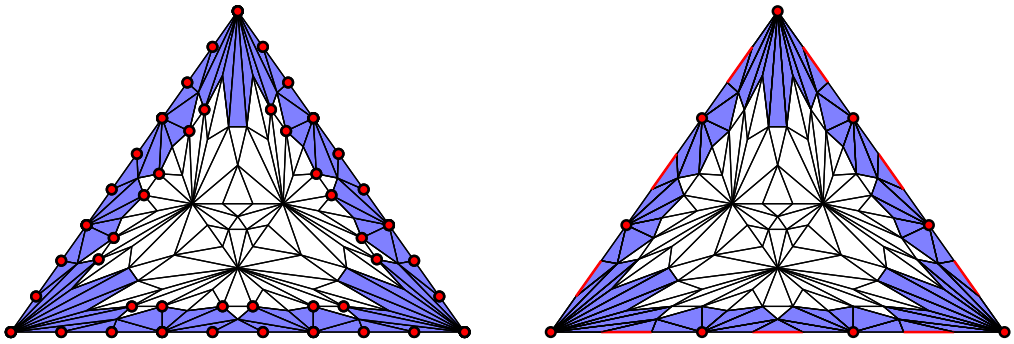
Proof. By Lemma 5, the affine task on $\mathcal{R}_k$ is solvable in the k-concurrency model. Moreover, according to Lemma 4, any task solvable in the k-concurrency model can be solved in the k-set consensus model. Therefore, by iterating a solution to the affine task $\mathcal{R}_k$, a run of $\mathcal{R}_k^*$ can be simulated in the k-set-consensus model and used to solve any task solvable in $\mathcal{R}_k^*$. ◀

4.3 From $\mathcal{R}_k^*$ to k-set consensus

Now we show how to simulate in $\mathcal{R}_k^*$ any algorithm that uses the AS memory and k-set-consensus objects.

***k*-set consensus simulation design.** A *non-blocking* simulation of the AS memory in $\mathcal{R}_k^*$ is straightforward, since the set of $\mathcal{R}_k^*$ runs is a subset of $(\mathrm{Chr}(\mathbf{s}))^*$ runs, and there exist several algorithms simulating the AS memory in $(\mathrm{Chr}(\mathbf{s}))^*$, e.g., [18].

Solving k-set consensus is not very complicated either in $\mathcal{R}_k^*$: every iteration of $\mathcal{R}_k$ provides a set of at most k *leaders*, i.e., processes with an IS^1output containing at most k elements, where at least one such *leader* is visible to every process, i.e., it can be identified as a *leader* and its input is visible to all. The set of leaders of $\mathcal{R}_2$ are shown in figure 4a in red, it is easy to observe that every simplex in $\mathcal{R}_2$ has at most two leaders, and that one is visible to every process (every process with a carrier of size at most 2 is a leader). This



(a) Leaders. **(b)** Processes with the smallest IS^2 output.

Figure 4 $\mathcal{R}_2$ for 3 processes: (a) leaders – vertices in red, and (b) processes with the smallest IS^2 output – simplices in red.

property gives a very simple k-set-consensus algorithm: every process decides on the value proposed by one of these k leaders. We will later show how this property can be derived from the restriction of $\mathcal{R}_k$ on the size of *contention sets*.

The real difficulty of the simulation consists in combining the shared-memory and k-set-consensus simulations, as in the simulated protocol, some processes may be accessing k-set-consensus objects while other processes are performing AS operations. Liveness of our k-set-consensus algorithm relies on the *participation* of visible leaders, i.e., on the fact that the leaders propose values for this instance of k-set-consensus. In this sense, our k-set-consensus algorithm may block if some leader is performing an AS operation or is involved in a different instance of k-set-consensus. Similarly if a "fast" process is involved in a k-set-consensus, then it can prevent every "slow" process to complete any AS memory operation as a write may be validated only after having been observed by every active process.

The solution we propose consists in (1) synchronizing the two simulations in order to ensure that, eventually, at least one process will complete its pending operation, and (2) ensuring that the processes collaborate by participating in every simulated operation. In our solution, every process tries to propagate every observed proposed value (for a write operation), and every process tries to reach a decision in every k-set-consensus object accessed by some process. For that, we make the processes participate in both simulation protocols (read-write and k-set-consensus) in every round of $\mathcal{R}_k^*$, until they decide.

Even though the simulated algorithm executes only one operation at a time and requires the output of the previous operation to compute the input for the following one, we enrich the simulated process with *dummy* operations that do not alter the simulation result. Then eventually some *undecided* process is guaranteed to complete both pending operations, where at most one of them is a *dummy* one. This scheme provides a *non-blocking* simulation of any algorithm using the AS memory and k-set-consensus objects.

We use the following observation. The shared memory simulation from [18] provides progress to the processes with the smallest snapshot output (i.e., with the smallest set of observed processes values). Our k-set-consensus algorithm provides progress to the leaders with the smallest $\mathcal{R}_k$ output, i.e., the processes with the smallest associated carrier. We synchronize the liveness properties of the two simulations by running the AS simulation only on every second round of the two rounds of restricted immediate snapshots associated to $\mathcal{R}_k$, denoted IS^2. For example, Figure 4b depicts the 2-dimensional of $\mathcal{R}_2$, where the sets of processes with the smallest IS^2 outputs are represented as red simplices.

Algorithm 4: k-set consensus simulation in $\mathcal{R}_k^*$: process i.

```
1  Init: r_i ← 0; State_i ← undecided; ConsId_i ← ⊥; ConsProp_i ← ⊥;
2  WriteVal_i[i] ← FirstWrite_i(); WriteCount_i[i] ← 1;
3  foreach m ∈ {1,...,n} \ {i} do (WriteCount_i[m], WriteVal_i[m]) ← (0, ⊥);
4  ConsHistory_i ← ∅ : List of adopted agreement proposals;

5  Repeat forever
6      r_i ← r_i + 1; Leaders_i ← true;
7      IS^2_output = R_k[r_i](State_i, (WriteCount_i, WriteVal_i), ConsHistory_i);

8      foreach (j, View_j) ∈ IS^2_output do
9          Let (State_j, (WriteCount_j, WriteVal_j), ConsHistory_j) ← RKInput(j);
10         foreach m ∈ {1,...,n} do
11             if WriteCount_j[m] > WriteCount_i[m] then
12                 WriteCount_i[m] = WriteCount_j[m], WriteVal_i[m] = WriteVal_j[m];
13         if |Undecided(View_j)| ≤ k then
14             if ∄(ConsId_i, *) ∈ ConsHistory_j then Leaders_i ← false;
15             foreach (A_id, A_val) ∈ ConsHistory_j do
16                 ReplaceOrAdd (A_id, *) in ConsHistory_i with (A_id, A_val);

17     if (Σ_{m∈{1,...,n}} WritesCount_i[m]) = r_i then
18         if PendingWriteSnapshotOperation() then
19             TerminateWriteOperation(WriteVal_i);
20         if Leaders_i ∧ ConsId_i ≠ ⊥ then
21             ConsProp_i ← A_val where (ConsId_i, A_val) ∈ ConsHistory_i;
22             TerminateAgreementOperation(A_val); ConsId_i ← ⊥;
23         if Terminated() then State ← decided;
24         else
25             WriteCount_i[i] ← WriteCount_i[i] + 1;
26             if NextAgreementOperation() = Available then
27                 (ConsId_i, ConsProp_i) ← NextAgreement_i();
28                 if (∄(A_id, A_val) ∈ ConsHistory_i with A_id = ConsId_i) then
29                     Add (ConsId_i, ConsProp_i) in ConsHistory_i;
30             if NextWriteSnapshotOperation() = Available then
31                 WriteVal_i[i] ← NextWrite_i();
32 End repeat;
```

This way at least the leader with the smallest $\mathcal{R}_k$ output will make progress in both simulations. Indeed, the definition of $\mathcal{R}_k$ implies that the set of processes with the smallest $\mathcal{R}_k$ outputs includes a leader, and a process with the smallest IS^2 output also has the smallest $\mathcal{R}_k$ output. Figure 4 gives an example of an intersection between the set of processes with the smallest IS^2 output and the set of leaders: here every process with the smallest IS^2 output has a carrier of size at most 2 and every such process is a leader.

***k*-set consensus simulation algorithm.** Algorithm 4 provides a simulation, in $\mathcal{R}_k^*$, of any protocol designed for the k-set-consensus model. The algorithm is based on the shared memory simulation from [18], applied on IS^2outputs of every iteration of $\mathcal{R}_k$, combined with a parallel execution of instances of our k-set-consensus algorithm. The simulation works in rounds that can be decomposed into three stages: communicating through $\mathcal{R}_k$, updating local information, and validating progress.

The first stage consists in accessing the new $\mathcal{R}_k$ iteration associated with the round, using information on the ongoing operations as an input (see line 7). For memory operations, an input to $\mathcal{R}_k$ consists of an array containing the most recent known *update* operation for every process, $WriteVal_i$, and the timestamp associated with the written value, $WriteCount_i$; $ConsHistory_i$ is a list of all adopted proposals for all accessed agreement operations. Finally, a value *State*, set to *decided* or *undecided*, is also put in $\mathcal{R}_k$'s input, to indicate whether the process has completed its simulation.

The second stage consists in updating the local information according to the output obtained from $\mathcal{R}_k$ (lines 8–16). The input value of each process observed in the second immediate snapshot of $\mathcal{R}_k$ is extracted (line 9). These selected inputs are examined in order to replace the local write values $WriteCount_i$ with the most recent ones, i.e., associated with the largest write counters (lines 10–12). The $ConsHistory$ variable of every leader, i.e., a process with an IS^1 output containing at most k *undecided* process inputs (using the variable *State*), is scanned in order to adopt all its decision estimates (lines 13–16). Moreover, the boolean value $Leaders_i$ is used to check if every observed leader transmitted a decision estimate for the pending agreement operation, $ConsId_i$.

The third stage consists in checking whether pending operations can safely be terminated (lines 17–22), and if so, whether the process has completed its simulation (line 23) or if new operations can be initiated (line 24–31).

Informally, it is safe for a process to decide in line 20, as there are at most k *Leaders* per round, one of which (1) is visible to every process and (2) provides a decision estimate for the pending agreement. Thus, every process adopts the decision estimate from a *leader* of the round, reducing the set of possible distinct decisions to k.

A pending memory operation terminates when the round number r_i equals the sum of the currently observed write counters (test at line 17), as in the original algorithm [18]. Indeed, the equality implies that the writes in the estimated snapshot have been observed by every process (line 19). Last, if a process did not terminate, it increments its write counter and, if there is a new operation available, the process selects the operation (see lines 25–31).

If there is a new agreement operation, then the input proposal and the object identifier are selected (line 27) and they are used for the current decision estimate in $ConsHistory_i$ (line 29), unless a value has already been adopted (line 28). If there is a new write operation then the current write value is simply changed (line 31), a *dummy* write thus consists in re-writing the same value.

▶ **Lemma 7.** *In $\mathcal{R}_k^*$, Algorithm 4 provides a non-blocking simulation of any algorithm designed for the k-set-consensus model.*

The proof of Lemma 7 is delegated to the companion technical report [15]. The main aspects of the proof are taken from the base algorithm from [18], while the liveness of the agreement operations relies on the restriction provided by $\mathcal{R}_k^*$ and the size of *contention sets*.

Lemma 7 implies the following result:

▶ **Theorem 8.** *Any task solvable in the k-set-consensus model can be solved in $\mathcal{R}_k^*$.*

Proof. To solve in $\mathcal{R}_k^*$ a task solvable in the k-set-consensus model, we can simply use Algorithm 4, simulating any given algorithm solving the task in the k-set-consensus model.

The non-blocking simulation provided by Algorithm 4 ensures, at each point, that at least one live process eventually terminates. As there are only finitely many processes, every live process eventually terminates. ◀

Lemma 4, Theorem 6, and Theorem 8 imply the following equivalence result:

▶ **Corollary 9.** *The k-concurrency model, the k-set-consensus model, and $\mathcal{R}_k^*$ are equivalent regarding task solvability.*

This equivalence result can be used to derived a generalization of the *asynchronous computability theorem* from [24] in its discrete formulation from what it means for a task to be solvable in $\mathcal{R}_k^*$:

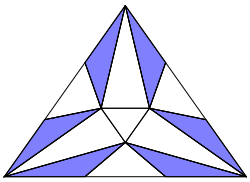


Figure 5 Fully ordered sub-Chr $\mathbf{s}$.

▶ **Theorem 10.** *Discrete k-Concurrent ACT: A task $(\mathcal{I}, \mathcal{O}, \Delta)$ is solvable in the k-concurrency or k-set-consensus model if and only if there exists a color-preserving, carrier-preserving, simplicial map ϕ: $\mathcal{R}_k^N \to \mathcal{O}$ for some natural number N.*

5 Concluding remarks: on minimality of $\mathrm{Chr}^2\,\mathbf{s}$ for k-set consensus

This paper shows that the models of k-set consensus and k-concurrency are captured by the same affine task $\mathcal{R}_k$, defined as a subcomplex of $\mathrm{Chr}^2\,\mathbf{s}$. One may wonder if there exists a simpler equivalent affine task, defined as a subcomplex of $\mathrm{Chr}\,\mathbf{s}$, the 1-degree of the standard chromatic subdivision. Just like for the t-resilient affine task [8, 32], this is in general not possible. Consider the case of $k = 1$ (consensus) in a 3-process system. We can immediately see that the corresponding subcomplex of $\mathrm{Chr}\,\mathbf{s}$ must contain all "ordered" simplexes depicted in Figure 5. Indeed, we must account for a wait-free 1-concurrent IS^1 run in which, say, p_1 runs first until it completes (and it must outputs its corner vertex in $\mathrm{Chr}\,\mathbf{s}$), then p_2 runs alone until it outputs its vertex in the interior of the face (p_1, p_2) and, finally, p_3 must output its interior vertex.

The derived complex is connected. Moreover, any number of its iterations still results in a connected complex. The simple connectivity argument implies that consensus cannot be solved in this iterated model and, thus, the complex cannot capture 1-concurrency.

Interestingly, the complex in Figure 5 precisely captures the model in which, instead of consensus, weaker *test-and-set* (TS) objects are used: (1) using TS, one easily make sure that at most one process terminates at an IS level, and (2) in IS runs defined by this subcomplex, any pair of processes can solve consensus using this complex and, thus, a TS object can be implemented. It is not difficult to generalize this observation to k-TS objects [27]: the corresponding complex consists of all simplices of $\mathrm{Chr}\,\mathbf{s}$, contention sets of which are of size at most k. The equivalence (requiring a simple generalization for the backward direction) can be found in [27, 20].

Overall, this raises an intriguing question whether every object, when used in the read-write system, can be captured via a subcomplex of $\mathrm{Chr}^m\,\mathbf{s}$ for some $m \in \mathbb{N}$.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

2 Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185–195, 2010.

3 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.

4 Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

5 Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC*, pages 189–198, 1997.

6 Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

7 Zohir Bouzid, Eli Gafni, and Petr Kuznetsov. Strong equivalence relations for iterated models. In *OPODIS*, pages 139–154, 2014.

8 Carole Delporte, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. t-resilient immediate snapshot is impossible. In *SIROCCO*, pages 177–191, 2016.

9 Pierre Fraigniaud, Eli Gafni, Sergio Rajsbaum, and Matthieu Roy. Automatically adjusting concurrency to the level of synchrony. In *DISC*, pages 1–15, 2014.

10 Eli Gafni. On the wait-free power of iterated-immediate-snapshots. Unpublished manuscript, `http://www.cs.ucla.edu/~eli/eli/wfiis.ps`, 1998.

11 Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.

12 Eli Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.

13 Eli Gafni and Rachid Guerraoui. Simulating few by many: Limited concurrency = set consensus. Unpublished manuscript, `http://web.cs.ucla.edu/~eli/eli/kconc.pdf`, 2009.

14 Eli Gafni and Rachid Guerraoui. Generalized universality. In *CONCUR*, pages 17–27, 2011.

15 Eli Gafni, Yuan He, Petr Kuznetsov, and Thibault Rieutord. Read-write memory and k-set consensus as an affine task. *arXiv preprint arXiv:1610.01423*, 2016.

16 Eli Gafni and Petr Kuznetsov. Relating l-resilience and wait-freedom via hitting sets. In *ICDCN*, pages 191–202, 2011.

17 Eli Gafni, Petr Kuznetsov, and Ciprian Manolescu. A generalized asynchronous computability theorem. In *PODC*, pages 222–231, 2014.

18 Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010.

19 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In *DISC*, pages 329–338, 2006.

20 Eli Gafni, Michel Raynal, and Corentin Travers. Test & set, adaptive renaming and set agreement: A guided visit to asynchronous computability. In *SRDS*, pages 93–102, 2007.

21 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

22 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2014.

23 Maurice Herlihy and Sergio Rajsbaum. The topology of shared-memory adversaries. In *PODC*, pages 105–113, 2010.

24 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

25 Damien Imbs, Sergio Rajsbaum, and Adrián Valle. Untangling partial agreement: Iterated x-consensus simulations. In *SSS*, pages 139–155, 2015.

26 Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(2):197–209, 2012.

27 Achour Mostéfaoui, Michel Raynal, and Corentin Travers. Exploring gafni's reduction land: From $Omega^k$ to wait-free adaptive (2p-[p/k])-renaming via k-set agreement. In *DISC*, pages 1–15, 2006.

28 Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The iterated restricted immediate snapshot model. In *COCOON*, pages 487–497, 2008.

29 Michel Raynal and Julien Stainer. Increasing the power of the iterated immediate snapshot model with failure detectors. In *SIROCCO*, pages 231–242, 2012.

30 Michel Raynal, Julien Stainer, and Gadi Taubenfeld. Distributed universality. In *OPODIS*, pages 469–484, 2014.

31 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

32 Vikram Saraph, Maurice Herlihy, and Eli Gafni. Asynchronous computability theorems for t-resilient systems. In *DISC*, pages 428–441, 2016.

33 Edwin H. Spanier. *Algebraic topology.* McGraw-Hill Book Co., New York, 1966.

Set-Consensus Collections are Decidable

Carole Delporte-Gallet[1], Hugues Fauconnier[2], Eli Gafni[3], and Petr Kuznetsov[*4]

1 **IRIF, Université Paris-Diderot, Paris, France**
cd@liafa.jussieu.fr
2 **IRIF, Université Paris-Diderot, Paris, France**
hf@liafa.jussieu.fr
3 **UCLA, Los Angeles, CA, USA**
eli@ucla.edu
4 **Télécom ParisTech, Paris, France**
petr.kuznetsov@telecom-paristech.fr

Abstract

A natural way to measure the power of a distributed-computing model is to characterize the set of tasks that can be solved in it. In general, however, the question of whether a given task can be solved in a given model is undecidable, even if we only consider the wait-free shared-memory model. In this paper, we address this question for restricted classes of models and tasks. We show that the question of whether a collection C of (ℓ, j)-*set consensus* objects, for various ℓ (the number of processes that can invoke the object) and j (the number of distinct outputs the object returns), can be used by n processes to solve wait-free k-set consensus is decidable. Moreover, we provide a simple $O(n^2)$ decision algorithm, based on a dynamic programming solution to the Knapsack optimization problem. We then present an *adaptive* wait-free set-consensus algorithm that, for each set of participating processes, achieves the best level of agreement that is possible to achieve using C. Overall, this gives us a complete characterization of a read-write model defined by a collection of set-consensus objects through its *set-consensus power*.

1998 ACM Subject Classification E.1.2 Distributed Data Structures, C.2.4 Distributed Systems

Keywords and phrases Decidability, distributed tasks, set consensus, Knapsack problem

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.7

1 Introduction

A plethora of models of computation were proposed for distributed environments. The models vary in timing assumptions they make, types of failures they assume, and communication primitives they employ. It is hard to say *a priori* whether one model provides more power to the programmer than the other. A natural way to measure this power is to characterize the set of distributed tasks that can be solved in a model. In general, however, the question of whether a given task can be solved in the popular *wait-free* read-write model, i.e., tolerating asynchrony and failures of arbitrary subsets of processes, is undecidable [13]. Of course, in models in which processes can additionally access arbitrary objects, the question is not decidable either. However, many natural models have been shown to be characterized by their power to solve set consensus [10].

* The author was supported by the Agence Nationale de la Recherche, under grant agreement N ANR-14-CE35-0010-01, project DISCMAT.

In this paper, we consider models in which n completely asynchronous processes communicate through reads and writes in the shared memory and, in addition, can access *set-consensus* objects. An (ℓ, j)-set-consensus object solves j-set consensus among ℓ processes, i.e., the object can be accessed by up to ℓ processes with *propose* operations that take natural numbers as inputs and return natural numbers as outputs, so that the set of outputs is a subset of inputs of size at most j. Set consensus is a generalization of consensus and, like consensus [18], exhibits a *universailty* property: ℓ processes can use (ℓ, j)-set consensus and read-write registers to implement j state machines, ensuring that at least one of them makes progress [12]. In this paper, we explore what level of agreement, and thus "degree of universality", can be achieved using any number of objects from a given set-consensus collection.

The special case when only one type of set consensus can be used in the implementation was resolved in [4, 8, 23]. Assuming that $k \geq j\lceil n/\ell \rceil$, we trivially solve $j\lceil n/\ell \rceil$-set consensus, by splitting n processes into $\lceil n/\ell \rceil$ groups of size ℓ (or less). A slightly more complex converse bound [4, 8, 23], accounting for the "delta" between n and $\ell\lceil n/\ell \rceil$, resolves the special case when only one type of set consensus object can be used.

Characterizing a general model in which processes communicate via objects in an arbitrary collection C of possibly different set-consensus objects is more difficult. For example, let C be $\{(2,1),(5,2)\}$, i.e., every 2 processes in our system can solve consensus and every 5 can solve 2-set consensus. What is the best level of agreement we can achieve using registers and an arbitrary number of objects in C in a system of 9 processes? One can easily see that 4-set consensus can be solved: the first two pairs of processes solve consensus and the remaining 5 invoke 2-set consensus, which would give at most 4 different outputs. One can also let the groups of the first 5 and the remaining 4 each solve 2-set consensus. (In general, any two set-consensus objects (ℓ_1, j_1) and (ℓ_2, j_2) can be used to solve $(\ell_1 + \ell_2, j_1 + j_2)$-set consensus.) But could we do $(9, 3)$-set consensus with C?

We propose a simple way to characterize the power of a set-consensus collection. By convention, let (ℓ_0, j_0) be $(1, 1)$, and note that $(1, 1)$-set consensus is trivially solvable. We show that a collection $C = \{(\ell_0, j_0), (\ell_1, j_1), \ldots, (\ell_m, j_m)\}$ solves (n, k)-set consensus if and only if there exist $x_0, x_1, \ldots, x_m \in \mathbb{N}$, such that $\sum_i \ell_i x_i \geq n$ and $\sum_i j_i x_i \leq k$. Thus, determining the power of C is equivalent to solving a variation of the Knapsack optimization problem [21], where each j_i serves as the "weight" of an element in C, i.e., how much disagreement it may incur, and each ℓ_i serves as its "value", i.e., how many processes it is able to synchronize. We describe a simple $O(n^2)$ algorithm for computing the power of C for solving set consensus among n processes using the dynamic programming approach.

The sufficiency of the condition is immediate. Indeed, the condition implies that we can partition the set of n processes in $\sum_i x_i$ groups: x_0 groups of size (at most) ℓ_0, x_1 groups of size (at most) ℓ_1, ..., x_m groups of size (at most) ℓ_m. Each of the x_i groups of size ℓ_i, $i = 0, \ldots, m$, can independently solve j_i-set consensus using a distinct (ℓ_i, j_i)-set-consensus object in C, which gives us at most $\sum_i j_i x_i \leq k$ different outputs in total.

The necessity uses a generalized version of the BG simulation [3, 5] that allows to simulate, in the read-write shared-memory model, a protocol that uses various types of set-consensus objects. We use this simulation to show that if a collection not satisfying the condition solves (n, k)-set consensus, then $k + 1$ processes can solve k-set consensus using read-write registers, contradicting the classical wait-free set-consensus impossibility result [3, 20, 24]. Interestingly, the necessity of this condition holds even if we can use read-write registers in addition to the elements in C.

Thus, we derive a complete characterization of models defined by collections of set-consensus objects. In particular, it allows us to determine the *j-set-consensus number* of a set-consensus collection C as the maximal number of processes that can achieve j-set consensus using C and read-write registers. Applied to arbitrary objects, this metric is a natural generalization of Herlihy's consensus number [18].

Coming back to the collection $C = \{(2, 1), (5, 2)\}$, our characterization implies that 4 is the best level of set consensus that can be achieved by 9 processes with C. Observe, however, that if only 2 processes participate, then they can use C to solve consensus, i.e., to achieve "perfect" agreement. Applying our condition, we also see that participating sets of sizes 3 up to 5 can solve 2-set consensus, participating sets of size 6 up to 7 can solve 3-set consensus, participating sets of size 8 up to 10 can solve 4-set consensus, etc. That is, for every given participating set, we can devise an *optimal* set-consensus algorithm that ensures the best level of agreement achievable with C.

An immediate question is whether we could *adapt* to the participation level and ensure the best possible level of agreement in any case? Such algorithms are very useful in large-scale systems with bounded contention levels. We show that this is possible by presenting an *optimally adaptive* set-consensus algorithm. Intuitively, for the currently observed participation, our algorithm employs the best algorithm and, in case the participating set grows, seamlessly relaxes the agreement guarantees by switching to a possibly less precise algorithm when there is a larger set of participants.

Our results thus imply that there is an efficient algorithm to decide whether one model defined by a collection of set-consensus object types can be implemented in model defined by another collection of set-consensus objects. We conjecture that the ability of any "reasonable" (yet to be defined precisely) shared-memory system to solve set consensus, captured by its j-set-consensus numbers, for all positive j, characterizes precisely its computing power with respect to solving tasks or implementing deterministic objects.

This work contributes to the idea that there is nothing special about consensus that set consensus cannot do. Indeed, set-consensus collections are decidable in the same way collections of consensus objects are [18]: the power of a collection of consensus objects $\{(\ell_1, 1), (\ell_2, 1), \ldots, (\ell_m, 1)\}$ to solve consensus is determined by $\max_i \ell_i$. Furthermore, it was recently shown that the computational power of a class of deterministic objects cannot be characterized by its ability to solve consensus [2], which suggests the use of set consensus in a characterization. We see this paper as the first step towards proving the conjecture that the computational power of a deterministic object can be captured by its *set-consensus* number, determining the best level of agreement the object can reach for each given system size.

Roadmap. The rest of the paper is organized as follows. In Section 2, we recall the basic model definitions and simulation tools. In Section 3, we present and prove our characterization of set-consensus collections, and describe an efficient algorithm to compute the characterizing criterion. In Section 4, we present an adaptive algorithm that achieves the optimal level of agreement for each set of active participants having access to a given set-consensus collection. We discuss related work in Section 5 and conclude in Section 6.

2 Preliminaries

In this section, we briefly state our system model, recall the notion of a distributed task, and sketch the basic simulation tools that we use in the paper.

Processes and tasks. We consider a system Π of asynchronous processes that communicate via shared memory abstractions. We assume that process may only fail by crashing, and otherwise it must respect the algorithm it is given. A *correct* process never crashes. Shared abstractions we consider here include an *atomic-snapshot* memory [1] and a collection of objects solving *distributed tasks* [20].

An atomic-snapshot memory stores a vector of $|\Pi|$ values, one value per process, and exports atomic operations *update* and *snapshot*: operation $update(p, v)$ performed by process p writes v in position p in the vector, and operation $snapshot()$ returns the vector. Atomic-snapshot memory can be implemented, in a wait-free and linearizable manner, in the standard read-write shared-memory model [1].

A process invokes a task with an input value and the task returns an output value, so that the inputs and the outputs across the processes invoked the task respect the task specification and every correct process that participates decides (gets an output). More precisely, a *task* is defined through a set $\mathcal{I}$ of input vectors (one input value for each process), a set $\mathcal{O}$ of output vectors (one output value for each process), and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input vector with a set of possible output vectors. An input $\perp$ denotes a *non-participating* process and an output value $\perp$ denotes an *undecided* process.

For vectors S and S' in $\mathcal{I}$ (resp., $\mathcal{O}$), we write $S \geq S'$ if S' is obtained from S by replacing some entries with $\perp$. We assume that if $\mathcal{I}$ (resp., $\mathcal{O}$) contains a vector S, then $\mathcal{I}$ (resp., $\mathcal{O}$) also contains any vector S' such that $S \geq S'$. We stipulate that if $(I, O) \in \Delta$, then (1) for all i, if $I[i] = \perp$, then $O[i] = \perp$, (2) for each O', such that $O \geq O'$, $(I, O') \in \Delta$ and, (3) for each I' such that $I' \geq I$, there exists some O' such that $O' \geq O$ for all i, if $I'[i] \neq \perp$, then $O'[i] \neq \perp$, and (I', O') in Δ.

An algorithm *solves a task* $T = (\mathcal{I}, \mathcal{O}, \Delta)$ *in a wait-free manner* if it ensures that in every execution in which processes start with an input vector $I \in \mathcal{I}$, every correct process decides, and the set of decided values, taken together with the processes taking these decisions, form a vector $O \in \mathcal{O}$ (where positions of non-decided processes are assigned $\perp$) such that $(I, O) \in \Delta$.

The task of k-set consensus. In the task of *k-set consensus*, input values are in a set of values V ($|V| \geq k + 1$), output values are also in V, and for each input vector I and output vector O, $(I, O) \in \Delta$ if the set of non-$\perp$ values in O is a subset of values in I of size at most k. The special case of 1-set consensus is called *consensus* [11]. More generally, *(ℓ, k)-set-consensus objects* ($k \leq \ell$) allow arbitrary subset of ℓ processes to solve k-set consensus.

Note that k-set consensus is an example of a *colorless* task (also known as a *convergence* task [5]): processes are free to use each others' input and output values, so the task can be defined in terms of input and output *sets* instead of vectors. Formally, let $val(U)$ denote the set of non-$\perp$ values in a vector U. In a colorless task, for all input vectors I and I' and all output vectors O and O', such that $(I, O) \in \Delta$, $val(I) \subseteq val(I')$ and $val(O') \subseteq val(O)$, we have $(I', O') \in \Delta$. To solve a colorless task, it is sufficient to find an algorithm that allows just one process to decide. Indeed, if such an algorithm exists, we can simply convert it into an algorithm that allows every correct process to decide: every process simply applies the decision function to the observed state of any process that has decided and adopts the decision.

In contrast, (ℓ, k)-set consensus is not colorless in a system of $n > \ell$ processes, as it does not always allow a process to adopt the decision of another process: e.g., if a process does not belong to a set S of ℓ processes, it cannot provide outputs for j-set consensus for S.

Simulation tools. An execution of a given algorithm $\mathcal{A}$ by the processes $p_1, \ldots p_n$ can be *simulated* by a set of *simulator* processes $s_1, \ldots, s_\ell$ (or, simply, simulators) that run a distributed algorithm "mimicking" the steps of $\mathcal{A}$ in a *consistent* way. Informally, for every execution E_s of the simulation algorithm, there exists an execution E of $\mathcal{A}$ by $p_1, \ldots, p_n$ such that the sequence of states simulated for every process p_i in E_s is observed by p_i in E.

A basic building block of our simulations is an *agreement* protocol [3, 5] that can be seen as a safe part of consensus. It exports one operation *propose*() taking $v \in V$ as a parameter and returning $w \in V$, where V is a (possibly infinite) *value set*. When a process p_i invokes *propose*(v) we say that p_i *proposes* v, and when the invocation returns v' we say that p_i *decides on* v'. Agreement ensures four properties:

(i) every decided value has been previously proposed,
(ii) no two processes decide on different values, and
(iii) if every participating process takes enough steps then eventually every correct participating process decides.

Here a process is called participating if it took at least one step in the computation. In fact, the agreement protocol in [3, 5] ensures that if every participating process takes at least three shared memory steps then eventually every correct participating process decides. If a participating process fails in the middle of an agreement protocol, then no process is guaranteed to return.

A generalized version of the agreement protocol, ℓ-*agreement* [4, 8], relaxes safety properties of agreement but improves liveness. Formally, in addition to (i) above, ℓ-agreement ensures:

(ii′) at most ℓ different values can be decided, and
(iii′) every correct participating process is guaranteed to decide, unless ℓ or more participating processes do not take enough steps.

Clearly, the agreement protocol we defined above is 1-agreement. An ℓ-agreement protocol with a proof (only sketched in [4, 8]) can be found in Reiners' thesis [23]. For completeness, given that the thesis is not easy to find, we present the proof in Appendix A.

3 A characterization of set-consensus collections

In this section, we introduce the notion of *agreement level* for a given set-consensus collection C and a given system size. Then we show that the metrics captures the power of C for solving set consensus. Then we show how to efficiently compute the agreement level of a given collection.

3.1 Agreement levels of C

Consider a model in which processes can communicate via an atomic-snapshot memory and set-consensus objects from a collection C. For brevity, we represent C as a set $\{(\ell_0, j_0), (\ell_1, j_1), \ldots, (\ell_m, j_m)\}$ such that for each $i = 0, \ldots, m$, the task of (ℓ_i, j_i)-set consensus can be solved $(\ell_i \geq j_i)$.

By convention, we assume that $(\ell_0, j_0) = (1, 1)$ is always contained in a collection C: $(1, 1)$-set consensus is trivially solvable. Note that (ℓ, j)-set consensus also solves (ℓ', j')-set consensus for all $\ell' \leq \ell$ and $j' \geq j$. Thus, without loss of generality, we can assume that the sequence $(\ell_1, j_1), \ldots, (\ell_m, j_m)$ is monotonically increasing: $\ell_0 < \ell_1$ and for all $i = 1, \ldots, m-1$, $\ell_i < \ell_{i+1}$ and $j_i < j_{i+1}$. (since we required that $(\ell_0, j_0) = (1, 1)$, there can be two elements of the type $(-, 1)$). In particular, for all n, C contains at most n elements (ℓ, j) such that $\ell \leq n$.

▶ **Definition 1** (Agreement level)**.** Let $C = \{(\ell_0, j_0), (\ell_1, j_1), \ldots, (\ell_m, j_m)\}$ be a collection of set-consensus objects. The *agreement level for n processes of C*, denoted AL_n^C, is defined as:

- $\min \sum_i j_i x_i$
- under the constraints: $\sum_i \ell_i x_i \geq n$, $x_0, \ldots, x_m \in \{0, \ldots, n\}$

One can also interpret AL_n^C as the lowest k for which there exists a *multiset* $S = \{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i = k$ and $\sum_i t_i \geq n$.[1]

3.2 Agreement levels and set consensus

We now can define a simple criterion to determine whether the model defined by C can solve (n, k)-set consensus. The criterion is *sufficient*, i.e., every model equipped with C that satisfies the criterion solves (n, k)-set consensus, and *necessary*, i.e., every model equipped with C that solves (n, k)-set consensus satisfies the criterion.

▶ **Theorem 2.** *(n, k)-set consensus can be solved using read-write registers and any number of objects taken in a set-consensus collection C if and only if $AL_n^C \leq k$.*

Proof. Suppose that $AL_n^C \leq k$. Thus, there exists a multiset $S = \{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i \leq k$ and $\sum_i t_i \geq n$. We show how n processes can solve k-set consensus using S. Every p_i, $i = 1, \ldots, n$, is assigned to the element $(t_j, s_j) \in S$ such that $\sum_{\ell=1,\ldots,j-1} t_\ell < i \leq \sum_{\ell=1,\ldots,j} t_\ell$, invokes the assigned object of (t_j, s_j)-set consensus with its input and returns the corresponding output. Since $\sum_i s_i \leq k$, the total number of outputs does not exceed k.

Now suppose that C can be used to solve (k, n)-set consensus and let A be the corresponding algorithm. By contradiction, suppose that no multiset S satisfying the conditions above exists for C. Thus, for any multiset $\{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i \leq k$, we have $\sum_i t_i < n$.

We show that we can then use a simulation of A to solve $(k + 1, k)$-set consensus using only read-write memory, contradicting the classical impossibility result [3, 20, 24]. The simulation we describe below is an extension of the *BG simulation* [3, 5], inspired by the algorithms described in [4, 8].

Simulation. Let $q_1, \ldots, q_{k+1}$ be a set of $k + 1$ simulator processes communicating via an atomic-snapshot memory. In its position in the snapshot memory, every simulator q_i maintains its estimate of the current simulated state of every simulated process in $\{p_1, \ldots, p_n\}$.

Note that the state of each p_ℓ (in algorithm A) unambiguously determines the next step that p_ℓ is going to take in the simulation, which can be an update operation, a snapshot operation, or an access to a (t, s)-set-consensus object. Since each update operation by p_ℓ is implicitly simulated by registering the last simulated state of p_ℓ in the shared memory, the simulators only need to explicitly simulate snapshot operations and accesses to set-consensus objects.

We associate each state of p_ℓ (assuming distinct local states) with a distinct agreement protocol (cf. Section 2), depending on the next step p_ℓ is going to take in that state:

- For a snapshot operation, we use one instance of the agreement (1-agreement) algorithm.
- For an access to a (t, s)-set-consensus object, we use one instance of s-agreement and one instance of 1-agreement.

[1] Note that assuming that $(1, 1) \in C$ implies $AL_n^C \leq n$.

The initial state of each simulated process is associated with a 1-agreement protocol.

The simulation proceeds in asynchronous rounds. In each round, a simulator q_i picks up the next simulated process p_ℓ in a round-robin fashion. To simulate a step of p_ℓ, q_i takes a snapshot of the memory and computes p_ℓ's latest simulated state by choosing the latest simulated state of p_ℓ found in the snapshot.

If p_ℓ is in the initial state, q_i invokes the agreement protocol (1-agreement) to compute the input of p_ℓ in the simulated run, using its input value (for k-set consensus) as a proposed value. Otherwise, q_i invokes the corresponding agreement protocol:

- To simulate a snapshot operation, q_i invokes the corresponding 1-agreement protocol, proposing the just read simulated system state (the vector of the latest simulated states of processes $p_1, \ldots, p_n$) as the outcome of the simulated snapshot.

  Recall that a simulator that has started but not finished the 1-agreement protocol for a given snapshot operation may block the simulated process forever. However, since the faulty simulator may be involved in at most one agreement protocol at a time, it can block at most one simulated process.
- To simulate an access of a (t, s)-set-consensus object, the simulator invokes the corresponding s-agreement protocol proposing p_ℓ's input value for this object (according to the simulated state) as the decided value.

  Recall that an s-agreement protocol may block forever if s or more processes fail in the middle of its execution. Thus, when it is used to simulate an access to (t, s)-set consensus, failures of s or more simulators may block t simulated processes.

  Also, recall that s-agreement may return different values to different simulators (as long as there are at most s of them). To ensure that the outcome of each of the t simulated processes accessing the (t, s)-set-consensus object is determined consistently by different simulators, the outcome of the simulated step is then agreed upon using 1-agreement.

If an agreement protocol for process p_ℓ blocks, simulator q_i proceeds to the next non-blocked simulated process in the round-robin order. If the corresponding agreement protocol terminates, the simulator updates the atomic-snapshot memory with its estimation of the simulated states of $p_1, \ldots, p_n$, where the new state of p_i is based on the outcome of the agreement.

Correctness. The use of 1-agreement protocols for both kinds of simulated operations implies that every step is simulated consistently, i.e., the simulators agree on the next simulated state of each process in $\{p_1, \ldots, p_n\}$.

The proposal to each of these agreement protocols is either the recently taken snapshot of the simulated system state (in case a snapshot operation is simulated) or the value that the simulated process must propose based on its state (in case an access to a (t, s)-set-consensus object is simulated). The initial state of each simulated process is an (agreed upon) input value of a participating simulator.

Each simulated snapshot is computed based on the most recent simulated states of $p_1, \ldots, p_n$ contained in the snapshot taken by the simulator "winning" the corresponding 1-agreement. The use of s-agreement in simulating accesses to a (t, s)-set-consensus object ensures that the simulated accesses return at most s proposed values. Thus, starting from the initial states of the simulated processes, we inductively derive that all states that appear in the simulated run are *compliant* with a run E of A: in E, each process p_i goes through the sequence of states that are agreed upon for p_i in the simulation.

Progress. It remains to show that at least one process in $\{p_1, \ldots, p_n\}$ *makes progress* in the simulated run, assuming that at least one of the $k+1$ simulators is correct. Consider any simulated run. We show that in this run, at least one of the simulated processes takes sufficiently many simulated steps (for producing an output for k-set consensus).

A simulated process may stop making progress only if an agreement protocol used for simulating its step blocks, which may happen only if a certain number of simulators stopped taking steps in the middle of the protocol.

Suppose that at most k simulators are faulty. Given that a faulty simulator can block at most one agreement protocol, we can identify the set of distinct agreement protocols $A_1 \ldots, A_p$ that are blocked in our run, and for all $j = 1, \ldots, p$, let A_i be s_i-agreement. We also identify p subsets of k faulty simulators of sizes $s_1, \ldots, s_p$, where s_i is the number of simulators that block A_i.

For each $i = 1, \ldots, p$, let t_i denote the number of simulated processes that are blocked because of A_i. If A_i is an instance of 1-agreement ($s_i = 1$) used to simulate a snapshot operation, to agree on the input of a given process, or to agree on the output of a set-consensus object at a given process, then we set $t_i = 1$ (only the corresponding simulated process can be blocked). Otherwise, A_i is an instance of s_i-agreement used to simulate an access to some (f_i, s_i)-set consensus, and we set $t_i = f_i$ (up to f_i processes accessing the (f_i, s_i)-set-consensus object can be blocked).

Since there are at most k faulty simulators, we get a multiset $\{(t_1, s_1), \ldots, (t_p, s_p)\}$ of elements in C such that $\sum_i s_i \leq k$. But then, by our contradiction hypothesis, we have $\sum_i t_i < n$, i.e., the total number of blocked simulated processes is less than n. Thus, at least one of the n processes $p_1, \ldots, p_n$ makes progress in the simulated run and eventually decides. Assuming that the first simulator to witness a decision in the simulated run writes it in the shared memory, we derive that every correct simulator eventually reads some decided value and decides.

Since all these values are coming from a run of an algorithm solving (n, k)-set consensus, there are at most k distinct decided values. Each of the decided values is an input of some simulator. Thus, $k+1$ simulators solve k-set-consensus using reads and writes – a contradiction. ◀

3.3 Computing the power of set-consensus collections

Having characterized the power of a collection C to solve set-consensus, we are now faced with the question of how to compute this power.

By Theorem 2, determining the best level of agreement that can be achieved by $C = \{(\ell_0, j_0), \ldots, (\ell_m, j_m)\}$ in a system of n processes is equivalent to finding $\min \sum_i j_i x_i$, under the constraints: $\sum_i \ell_i x_i \geq n$, $x_0, x_1, \ldots, x_m \in \{0, \ldots, n\}$. This can be viewed as a variation of the Knapsack optimization problem [21], where we aim at minimizing the total weight of a set of items from C put in a knapsack, while maintaing a predefined minimal total value of the knapsack content.[2] Here each j_i serves as the "weight" of an element in C, i.e., how much disagreement it may incur, and each ℓ_i serves as its "value", i.e., how many processes it is able to synchronize. We use this observation to derive an algorithm to compute AL_n^C in $O(n^2)$ steps.

Recall that C is represented as a monotonically increasing sequence $(\ell_0, j_0), \ldots, (\ell_m, j_m)$.

[2] The classical Knapsack optimization problem consists in maximizing the total value, while maintaining the total weight within a given bound.

First we *complete* C for the fixed system size n: for each $i = 1, \ldots, m$, such that $j_i < n$, we insert elements $(\max(j_i+1, \ell_{i-1}+1), j_i)$,$(\max(j_i+1, \ell_{i-1}+1)+1, j_i)$, ..., $(\min(\ell_i, n)-1, j_i)$, $(\min(\ell_i, n), j_i)$. For example, the completion of $C = \{(1,1), (3,2), (10,6)\}$ for $n = 11$ would be $\{(1,1), (3,2), (7,6), (8,6), (9,6), (10,6)\}$. Notice that since (ℓ_0, j_0), ..., (ℓ_m, j_m) is monotonically increasing, such a completion can be performed in $O(n)$ steps, and the resulting sequence is also monotonically increasing.

As a result of the completion, for every $r = 1, \ldots, n$, and each element of the kind $(\ell, j) \in C$ such that $\ell > r$ and $j < r$ we have a new element (r, j). As we will see below, this allows us to compute AL^C_r in $O(r^2)$ steps.

We observe that for all $r = 1, \ldots, n$, $AL^C_r = \min_{\ell_i \leq r}(j_i + AL^C_{r-\ell_i})$. Indeed, for all (ℓ_i, j_i) such that $\ell_i \leq r$, it must hold that $j_i + AL^C_{r-\ell_i} \geq AL_r$, otherwise, (ℓ_i, j_i) plus the multiset $(t_1, s_1), \ldots, (t_p, s_p)$ of elements in C that reaches $AL_{r-\ell_i}$ would give $j_i + \sum_v s_v < AL^C_r$ and $\ell_i + \sum_v t_v \geq \ell_i + r - \ell_i = r$, contradicting the definition of AL^C_r. Further, since C is complete, for each multiset $(t_1, s_1), \ldots, (t_p, s_p)$ in C reaching AL^C_r, we can construct a multiset $(\min(t_1, r), t_1), \ldots, (\min(t_p, r), s)$ in C (each set-consensus object in the multiset is defined for at most r processes) that also reaches AL^C_r. Hence, $AL^C_r = \min_{\ell_i \leq r}(j_i + AL^C_{r-\ell_i})$

Thus, we can use the following simple iterative algorithm (a variant of a solution to the Knapsack optimization problem based on dynamic programming) to compute AL^C_n in $O(n^2)$ steps:

$AL^C_0 = 0$;
for $r = 1, \ldots, n$ **do** $AL^C_r = \min_{\ell_i \leq r}(j_i + AL^C_{r-\ell_i})$.

In each iteration $r = 1, \ldots, n$ of the algorithm above, we perform at most r checks, which gives us $O(n^2)$ total complexity.

We can also consider a related notion of *j-set-consensus number* of C, denoted SCN^C_j and defined as the maximal number of processes that can achieve j-set consensus using C and read-write registers: $SCN^C_j = \max_{AL^C_n \leq j} n$. This is a natural generalization of Herlihy's consensus power [18]. Note that the problem of computing SCN^C_j is the classical Knapsack optimization problem, and using a variation of the algorithm above we can do it in $O(j|C|)$ steps (see, e.g., [21, Chap. 5]).

4 An adaptive algorithm: reaching optimal agreement

Theorem 2 implies that, for every fixed n, there exists an AL^C_n-set-agreement algorithm $\mathcal{ST}^C_n$ ($\mathcal{ST}$ for *static*) using C. We show that these algorithms can be used in an *adaptive* manner, so that for each set of participating processes, the best possible level of agreement can be achieved.

To understand the difficulty of finding such an adaptive algorithm, consider $C = \{(1,1), (13,5), (20,9)\}$. For selected sizes of participating sets m, the table in Figure 1 gives AL^C_m, and lists the elements of C used in the corresponding $\mathcal{ST}^C_m$.

If we have 16 processes, $\mathcal{ST}^C_{16}$ uses one instance of $(13,5)$-set consensus and three instances of $(1,1)$-set consensus to achieve $AL^C_{16} = 9$. But if two new processes arrive we need $(20,9)$-set consensus to achieve $AL^C_{18} = 9$. Interestingly, to achieve $AL^C_{22} = 10$, we should abandon $(20,9)$-set consensus and use two instances of $(13,5)$-set consensus instead. In other words, we cannot simply add a set-consensus instance of the species we used before to account for the arrival of new processes. Instead, we have to introduce a new species.

We present a wait-free adaptive algorithm that ensures that if the set of participating processes is of size m, then at most AL^C_m distinct input values can be output. We call such an algorithm *optimally adaptive for C*.

m	AL_m^C	$\mathcal{ST}_m^C$
1	1	(1,1)
2	2	(1,1) (1,1)
3	3	(1,1) (1,1) (1,1)
4	4	(1,1) (1,1) (1,1) (1,1)
5 to 13	5	(13,5)
14	6	(13,5) (1,1)
15	7	(13,5) (1,1) (1,1)
16	8	(13,5)(1,1)(1,1)(1,1)
17	9	(13,5)(1,1)(1,1)(1,1)(1,1) or (9,20)
18 to 20	9	(20,9)
21	10	(20,9)(20,9) or (13,5)(13,5)
22 to 26	10	(13,5) (13,5)
...		

Figure 1 Selecting elements in $C = \{(1,1), (13,5), (20,9)\}$ to solve AL_m^C-set consensus.

The algorithm is presented in Figure 2. The idea is the following: periodically, every process p writes its current value (initially, its input), together with the number of processes it has seen participating so far (initially, 0) in the shared memory, and takes a snapshot to get the current set P of participating processes and their inputs.

Process p then computes its *rank* in P and adopts the value v from a process announcing the largest participating set. The chosen input is then proposed to an instance of algorithm $\mathcal{ST}_{|P|}^C$, where p behaves as the process at position *rank* and proposes value v. More precisely, $\mathcal{ST}_{|P|}^C$ is treated as an algorithm for processes $q_1, \ldots, q_{|P|}$ and, thus, p runs the code of process q_{rank} in the algorithm with input value v.

Note that since the set is derived from an atomic snapshot of the memory, the notion of the largest participating set is well-defined: the snapshots of the same size are identical. Therefore, at most $|P|$ processes participate in $\mathcal{ST}_{|P|}^C$ and each of these $|P|$ processes can only participate at a distinct position corresponding to its rank in P. As a result, every correct process invoking $\mathcal{ST}_{|P|}^C$ will eventually get an output of the "best" set-consensus algorithm for P.

When the participating set P observed by p does not change in two consecutive iterations, p terminates with its current value.

▶ **Theorem 3.** *Let $\mathcal{C}$ be a set-consensus collection, n be an integer. The algorithm in Figure 2 is optimally adaptive for $\mathcal{C}$ in a system of n processes.*

Proof. We show first that every correct process eventually returns a value, and any returned value is a proposed one.

Let p and q take snapshots (Lines 2 or 11) in that order, and let P_p and P_q be, respectively, the returned participating sets. We observe first that $P_p \subseteq P_q$. Indeed, each position in the snapshot object R is initialized to $(\perp, \perp)$. Once, p updates $R[p]$ with its value and participation level, the position remains non-$\perp$ forever. Thus, if q takes its snapshot of R after p, then P_p, the set of processes whose positions are non-$\perp$ in the resulting vector, is a subset of P_q.

Therefore, the sets P and *parts* evaluated by p in Line 11 are non-decreasing with time. Since $\mathcal{ST}_{|parts|}^C$ is wait-free, the only reason for a correct process p not to return is to find that $parts \subsetneq P$ in Line 13 infinitely often, i.e., both P and *parts* grow indefinitely. But the two sets are bounded by the set Π of all processes – a contradiction.

Furthermore, every returned value is a value decided in an instance of $\mathcal{ST}_{|parts|}^C$. But every value proposed to algorithm $\mathcal{ST}_{|parts|}^C$ was previously read in a non-$(\perp, \perp)$ position of R, which can only contain an input value of some process.

Hence, every correct process eventually returns a value, and any returned value is a proposed one.

Shared objects:

R: `snapshot object, storing pairs` $(value, level)$, `initialized to` $(\perp, \perp)$

Local variables for process $p \in \Pi$:

$r[1, \ldots, n]$: `array of pairs` $(value, level)$
$prop, v$: $value$
$parts, P \in 2^{\Pi}$
$index$: integer

Code for process $p \in \Pi$ with proposal v_p:

1 $R.update(p, (v_p, 0))$
2 $r[1, \ldots, n] = R.snapshot()$
3 P=set of processes q such that $r[q] \neq (\perp, \perp)$
4 **repeat**
5 $parts := P$
6 $rank :=$ the rank of p in $parts$
7 $k :=$ be the greatest integer such that $(-, k)$ is in r
8 $v :=$ be any value such that (v, k) is in r
9 $prop := \mathcal{ST}^C_{|parts|}$ with value v at position $rank$
10 $R.update(p, (prop, |parts|))$
11 $r[1, \ldots, n] = R.snapshot()$
12 P=set of processes q such that $r[q] \neq (\perp, \perp)$
13 **until** $parts = P$
14 **return** $prop$

Figure 2 An optimally adaptive set-consensus algorithm.

Now consider a run of the algorithm in Figure 2 in which m processes participate. We say that a process p *returns at level* t in this run if it outputs (in Line 14) the value $prop$ returned by the preceding invocation of $\mathcal{ST}^C_t$ (in Line 9). By the algorithm, if p returns at level t, then the set $parts$ of processes it witnessed participating is of size t.

Let ℓ be the smallest level ($1 \leq \ell \leq n$) at which some process returns, and let O_ℓ be the set of values ever written in R at level ℓ, i.e., all values v, such that (v, ℓ) appears in R.

We show first that for all $\ell' > \ell$, if R contains (v', ℓ'), then $v' \in O_\ell$.

By contradiction, suppose that some process q is the first process to write a value (v', ℓ') (in Line 10), such that $\ell' > \ell$ and $v' \notin O_\ell$, in R. Thus, the immediately preceding snapshot taken by q before this write (in Lines 2 or 11) witnessed a participating set of size ℓ'. Hence, the snapshot of q occurs after the last snapshot (of size $\ell < \ell'$) taken by any process p that returned at level ℓ. But immediately before taking its last snapshot, every such process p has written (v, ℓ) in R (Line 10) for some $v \in O_\ell$. Thus q must see (v, ℓ) in its snapshot of size ℓ' and, since, by the assumption, the snapshot contains no values written at levels higher than ℓ, q must adopt some value written at level ℓ (Lines 7 and 8). Thus, $v' \in O_\ell$ – a contradiction.

Thus, every returned value must appear in O_ℓ, where ℓ is the smallest level ($1 \leq \ell \leq n$) at which some process returns. Now we show that $|O_\ell| \leq AL^C_m$, recall that m is the number of participating processes.

Indeed, since all values that appear in O_ℓ were previously returned by the algorithm $\mathcal{ST}^C_\ell$ (Line 9) and, as we observed earlier, the algorithm is used by at most ℓ processes, each choosing a unique position based on its rank in the corresponding snapshot of size ℓ, there can be at most AL^C_ℓ such values. Since at most m processes participate in the considered run, we have $\ell \leq m$, and, thus, $AL^C_\ell \leq AL^C_m$.

Hence, in a run with participating set of size m, $|O_\ell| \leq AL^C_m$ and, thus, at most AL^C_m values can be returned by the algorithm. Thus, we indeed have an optimally adaptive set-consensus algorithm using C. ◀

On unbounded concurrency. Our definitions of the agreement level and the set-consensus number of a set-consensus collection are independent of the size of the system: they are defined with respect to a given participation level. Our adaptive algorithm does account for the system size, as it uses atomic snapshots. But by employing the atomic-snapshot algorithms for unbounded-concurrency models described in [16], we can easily extend our adaptive solution to these models too.

5 Related work

Our algorithm computing the power of a set-consensus collection in $O(n^2)$ steps (for a system of n processes) is inspired by the dynamic programming solution to the Knapsack optimization problem described, e.g., in [21, Chap. 5].

Herlihy [18] introduced the notion of *consensus number* of a given object type, i.e., the maximum number of processes that can solve consensus using instances of the type and read-write registers. It has been shown that n-process consensus objects have consensus power n. However, the corresponding consensus hierarchy is in general not robust, i.e., there exist object types, each of consensus number 1 which, combined together, can be used to solve 2-process consensus [22]. Besides objects of the same consensus number m may not be equivalent in a system of more than m processes [2].

Borowsky and Gafni [4], and then Chaudhuri and Reiners [8, 23] independently explored the power of having multiple instances of (ℓ, j)-set-consensus objects in a system of n processes with respect to solving set consensus, which is a special case of the question considered in this paper. The characterization of [4, 8, 23] is established by a generalized BG simulation [3, 5] by Borowsky and Gafni, where instead of 1-agreement protocol, a more general j-agreement protocol is used. Our results employ this agreement protocol to show a more general result.

Gafni and Koutsoupias [13] and Herlihy and Rajsbaum [19] showed that wait-free solvability of tasks for 3 or more processes using registers is an undecidable question. We show that in a special case of solving set consensus using a set-consensus collection, the question is decidable. Moreover, we give an explicit polynomial algorithm for computing the power of a set-consensus collection.

6 Concluding remarks

We hope that this work will be a step towards proving a more general conjecture that our set-consensus numbers capture precisely the computing power of any "natural" shared-memory model. An indication that the conjecture is true is that set-consensus objects

are, in a precise sense, *universal* (generalizing the consensus universality [18]): using (n, k)-set-consensus objects, n processes can implement k independent sequential state machines so that at least one of them is able to make progress, i.e., to execute infinitely many commands [12]. Popular restrictions of the runs of the wait-free model, such as *adversaries* [10] and *failure detectors* [7, 6], were successfully characterized via their power for solving set consensus [14, 15, 9]. Also, it can be inferred from the recent result by Afek et al. [2] that, like consensus, set-consensus objects can express precisely certain deterministic objects [2]. We therefore believe that the power of a large class of "natural" models (determined by restrictions mentioned above) can be captured by their ability to solve set consensus. This class must exclude models in which "in between" objects, like *Weak Symmetry-Breaking* [20, 17], are used: such models, as we believe, cannot be expressed as a restriction of the runs of the wait-free model, and are therefore not "natural".

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

2 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *PODC*, 2016.

3 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100. ACM Press, May 1993.

4 Elizabeth Borowsky and Eli Gafni. The implication of the Borowsky-Gafni simulation on the set-consensus hierarchy. Technical report, UCLA, 1993. `http://fmdb.cs.ucla.edu/Treports/930021.pdf`.

5 Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

6 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

7 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

8 Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation (extended abstract). In *WDAG*, pages 362–379, 1996.

9 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Petr Kuznetsov. Wait-freedom with advice. *Distributed Computing*, 28(1):3–19, 2015.

10 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24(3-4):137–147, 2011.

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, April 1985.

12 Eli Gafni and Rachid Guerraoui. Generalized universality. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 17–27, Berlin, Heidelberg, 2011. Springer-Verlag.

13 Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM J. Comput.*, 28(3):970–983, 1999.

14 Eli Gafni and Petr Kuznetsov. Turning adversaries into friends: Simplified, made constructive, and extended. In *OPODIS*, pages 380–394, 2010.

15 Eli Gafni and Petr Kuznetsov. Relating L-Resilience and Wait-Freedom via Hitting Sets. In *ICDCN*, pages 191–202, 2011.

16 Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *PODC*, pages 161–169, 2001.

17 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In *DISC*, pages 329–338, 2006.

18 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

19 Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks (extended abstract). In *STOC*, pages 589–598, 1997.

20 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.

21 Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

22 Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000.

23 Paul Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. Master's thesis, Iowa State University, 1996.

24 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

A An ℓ-agreement algorithm

The algorithm (presented in Figure 3) uses two *atomic snapshot* objects A and B, initialized with $\perp$'s. A process writes its input in A (line 15) and takes a snapshot of A (line 16). Then the process writes the outcome of the snapshot in B (line 17) and keeps taking snapshots of B until it finds that at most $\ell - 1$ participating (i.e., having written their values in A) processes that have not finished the protocol, i.e., have not written their values in B (Lines 18-21). Finally, the process returns the smallest value (we assume that the value set is ordered) in the smallest-size non-$\perp$ snapshot found in B (containing the smallest number of non-$\perp$ values). (Recall that all snapshot outcomes are related by containment, so there indeed exists such a smallest snapshot.)

▶ **Theorem 4.** *The algorithm in Figure 3 implements ℓ-agreement.*

Proof. The validity property (i) is immediate: only the identifier of a participating process can be found in a snapshot object. The termination property (iii)$'$ of ℓ-agreement is immediate:

Shared objects:

A, B: `snapshot objects, initially` $\perp$

propose(v)

15 A.*update*(v)
16 $U := A$.*snapshot*()
17 B.*update*(U)
18 **repeat**
19 $\quad W := B$.*snapshot*()
20 $\quad X := \{j | (U[j] \neq \perp) \wedge (W[j] = \perp)\}$
21 **until** $|X| \leq \ell - 1$
22 $S :=$ the smallest-size set of non-$\perp$ values contained in $\{ W[j];\ j = 1, \ldots, n,\ W[j] \neq \perp\}$
23 *return* min(S)

Figure 3 The ℓ-agreement algorithm.

if at most $\ell - 1$ processes that have executed line 15 fail to execute line 17, then the exit condition of the repeat-until clause in line 21 eventually holds and every correct participating process terminates.

Suppose, by contradiction, that at least $\ell+1$ different values are returned by the algorithm. Thus, at least $\ell + 1$ distinct snapshots were written in B by $\ell + 1$ processes. Let L be the set of processes that have written the ℓ smallest snapshots in B in the run. The set is well-defined as all snapshots taken in A are related by containment. We are going to establish a contradiction by showing that every process must return the smallest value in one of the snapshots written by the processes in L and, thus, at most ℓ distinct inputs will be produced.

Let p_i be any process that completed line 17 by writing the result of its snapshot of A in B. Let U be the set of processes that p_i witnessed in A and, thus, wrote to its position in B in line 17.

If $p_i \in L$, i.e., U is one of the ℓ smallest snapshots ever written in B, then p_i will return the value of the smallest process in U or a smaller snapshot written by some process in L. If $p_i \notin L$, then U contains all ℓ distinct snapshots written by the processes in L. Since each process in L is included in the snapshot it has written in B, we derive that $L \subseteq U$. Since p_i returns a value only if all but at most $\ell - 1$ processes it witnessed participating have written their snapshots in B, at least one snapshot written by a process in L is read by p_i in B. Thus, p_i outputs the value of the smallest process in the snapshot written by a process in L – a contradiction.

Thus, at most ℓ distinct values can be output and (ii)$'$ is satisfied. ◀

k-Set Agreement in Communication Networks with Omission Faults

Emmanuel Godard[1] **and Eloi Perdereau**[2]

1 **Aix Marseille University, CNRS, LIF, Marseille, France**
emmanuel.godard@lif.univ-mrs.fr
2 **Aix Marseille University, CNRS, LIF, Marseille, France**
eloi.perdereau@lif.univ-mrs.fr

Abstract

We consider an arbitrary communication network G where at most f messages can be lost at each round, and consider the classical k-set agreement problem in this setting. We characterize exactly for which f the k-set agreement problem can be solved on G.

The case with $k = 1$, that is the Consensus problem, has first been introduced by Santoro and Widmayer in 1989 [20], the characterization is already known from [10]. As a first contribution, we present a detailed and complete characterization for the 2-set problem. The proof of the impossibility result uses topological methods. We introduce a new subdivision approach for these topological methods that is of independent interest.

In the second part, we show how to extend to the general case with $k \in \mathbb{N}$. This characterization is the first complete characterization for this kind of synchronous message passing model, a model that is a subclass of the family of oblivious message adversaries.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases k-set agreement, message passing, dynamic networks, message adversary, omission faults

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.8

1 Introduction

We consider communication networks with arbitrary topology where some messages can be lost. The system evolves in rounds, and at each round, at most f messages can be lost, the unreliable links can be different in each round. We consider the classical k-set agreement problem that has been introduced in 1993 by Chaudhuri [9]. The problem, for n processes, is defined informally as follows. Given $k + 1$ possible initial values, each process must decide a final value among the proposed values in such a way that there are at most k different decided values. Note that when $n = k + 1$, this problem is also defined as the set agreement problem.

The system can be modeled by an adversary that in every round can choose f messages to be "omitted". The corresponding faulty communication links are not necessarily the same at each round and can be changed later by the adversary, it is an oblivious adversary. Such faulty communication actually induces a sub-directed graph of G (or sub-digraph). Such directed graphs are called "instant graphs". The message adversaries terminology was introduced in [1] despite this model being introduced a long time ago, see Subsection 1.3. Since a general characterization of the solvability of the k-set agreement is still elusive for synchronous message passing (whereas the minimal failure detector to solve this problem is known for shared memory [11]), it is of interest to consider such special cases. This communication model is an important case of oblivious message adversaries.

1.1 Our Result

We give a complete characterization of the k-set agreement problem for networks with arbitrary topology in the omission faults model. We introduce a new combinatorial parameter for any graph G, denoted $c_k(G)$, that is the number of edge removals that G can withstand while having at most k connected components. This parameter is an extension of the classical edge-connectivity of a graph (that corresponds to $c_1(G)$ with our notation).

We prove that k-set agreement is solvable in G despite at most f message losses per round if and only if $f \leq c_k(G)$. The necessary condition is first proved for graphs of size $k+1$ using a reduction to the Sperner's lemma. For general graphs, we then show how to reduce to one of these graphs with $k+1$ vertices.

Interestingly, while the case of the $k+1$-clique corresponds to the standard chromatic subdivision found in topological proofs for distributed computability, we had to introduce a new technique, that is called a round diagram, in order to solve other non-complete topology cases. This new technique could be of independent interest in distributed computability.

1.2 Related Works

The k-set agreement problem is a classical paradigm of coordination problems. It is also a theoretical benchmark for distributed computability in numerous models. A recent review by Raynal can be found in [17].

The solvability of the 1-set agreement problem, that is, the Consensus problem, in the context of communication networks with arbitrary topology has been introduced by Santoro and Widmayer in 1989 [20, 21]. It has been fully characterized for arbitrary oblivious message adversaries in [10] answering the same problem as this paper for $k = 1$, a problem that was open since [20]. In this setting, the Consensus problem is equivalent to the Broadcast problem, that is the network should be connected at each round, i.e. f must be less than the connectivity of the underlying graph to be able to solve 1-set agreement.

The solvability of the k-set problem has been considered in the omission context by [12]. The communication graph is the complete graph and the omissions are counted in the whole execution. By contrast, our work present models that can withstand an infinite number of omissions.

The k-set agreement problem has been investigated in the context of dynamic networks in [2, 3], where the adversaries are non-oblivious. We have been recently made aware of an independent work of Biely et al, [4] under submission, that presents an algorithm that would solve k-set agreement in the sufficient condition of Section 5. Like in the $k = 1$ case, where Consensus algorithms are actually simple flooding algorithms, note that the main contribution here is for the impossibility results.

In the shared memory model, the impossibility of wait-free k-set agreement for more than $k+1$ processes is one of the crowning achievements of topological methods in distributed computing [15, 19, 6].

1.3 Related Models

The failure model considered here is very relevant in many ways. This model of synchronous communication has actually been introduced numerous times under different names. We present briefly the mobile omissions model [20] then the more recent "Heard-of" model [8], the iterated snapshot model [5] and its final evolution as the *message adversary* model [1]. Some equivalences have been proved between these synchronous presentations and asynchronous models in the case of non-coloured tasks [7]. Note also that in the case of dynamic networks,

whenever the communication primitive is a broadcast (to the current neighbours), this model can also be used.

Mobile Omissions / Omission Faults. This is the model originally used in [20] and [21] by Santoro and Widmayer. At a given moment, there are at most f omission faults, that is f arcs missing from the underlying graph G. In the following step, it is possible that omissions have "moved" at other arcs of the network. Hence the name. Note that in [20, 21], other kind of failures are also considered like byzantine failures. These are oblivious adversaries.

Heard-of Model. The "Heard-of" model has been presented by Charron-Bost and Schiper in [8] to model what the authors have called "benign faults", i.e. transient faults like omission faults. The presentation is mostly done in the logical form, where a special predicate HO describes for every node the set of nodes it received a message from in the current step. The system is evolving synchronously except that nodes do not start the algorithm at the same round. In this model, numerous families were considered, some are oblivious, other not.

Iterated Write Snapshot. This is a shared memory model. Single-writers/multi-readers registers are accessible by processes. There is usually as many registers as processes and the registers are arranged in a one-shot array. It can be assumed, as in [5], that there is a `writeSnapshot()` primitive that enables processes to atomically write values to their register and read the content of the full array. Each concurrent access can read the values corresponding to the calling process and also the values previously written by other processes in the same round. In a given round, all possible interleaving of calls to `writeSnapshot()` are allowed. Process may never fail, however, it is possible that a correct process never sees another correct process (e.g. if it always writes first).

The main interest of this model is that it has a simple synchronous and regular structure and that it was proved in [7] that a bounded colourless task can be wait-free solved in the classical read-write model if and only if it is solvable in the Iterated Write Snapshot model. So this model has the same computing power as shared memory, but using topological tools is simpler in this model (see the tutorial in [14] and the thorough coverage of [13]).

Message Adversaries. The communication structure that one gets with a shared memory model is usually edge-transitive. Considering message passing systems, this condition is actually not necessary, and in [1], where the terminology of "message adversaries" is introduced, this condition is dropped by considering various families of graphs where the instant graphs are not transitive.

In [1], Afek and Gafni show that the same tasks can be solved as in standard asynchronous shared memory if the instant digraphs are tournaments. So some message adversaries with specific non-complete topologies have the same computing power as classical shared memory.

Subsequently, Raynal and Stainer have shown in [18] that it is possible to consider various message adversaries (they are not all oblivious) where further restrictions on the set of possible scenarios, i.e. weaker adversaries, correspond to well known asynchronous shared-memory models enriched with failures detectors. It appears that the message adversary model is a very rich, but also very convenient (some have very simple protocol complex) model to describe distributed systems from the point of view of computability.

1.4 Outline

We first present our notations and formally define the communication model. Section 3 considers G to be one of the graphs K_3 and P_3 (the 3-clique and the 3-path), and using two

different subdivisions of the 2-dimensional simplex (one for each graph), we show that the number of possible failures is less than $c_2(G)$ for both cases. We show this is optimal for the set agreement problem. Then, in Section 4, we consider graphs of arbitrary size and show how to reduce the solvability of the 2-set agreement problem to one of the previous cases.

In the last section we investigate the general case $k \in \mathbb{N}$. We first show how to prove the impossibility of the set agreement problem for graphs of size $k+1$ using the chromatic subdivision technique with a twist, some instant graphs can appear more than one time in the subdivision. We then show that the reduction of the case $k = 2$ easily extends to arbitrary k and prove the necessary condition. Finally, a simple k-set agreement algorithm demonstrates that the condition $f \leq c_k(G)$ is sufficient, the characterization is complete.

2 Model and Definitions

2.1 Graphs and digraphs

Let $G = (V, E)$ be an undirected graph, we note $dir(E) = \bigcup_{\{u,v\} \in E} \{(u, v), (v, u)\}$ the set where each edge is replaced by two symmetric arcs ; we then have $|dir(E)| = 2|E|$. By extension of notation, for every undirected graph $G = (V, E)$, we note $dir(G) = (V, dir(E))$ the corresponding directed graph.

Given a graph G, a sub-digraph of $dir(G)$ is called an *instant graph*.

Let $D = (V, A)$ be a directed graph and $p \in V$ a vertex. The in-neighbourhood of each vertex $p \in V$ is denoted by $N_D^-(p)$ and corresponds to the set of sources of each arc reaching p in A:

$$\forall p \in V \quad N_D^-(p) = \{q \in V \mid (q, p) \in A\}$$

A directed path, or dipath, from p to q in D is a sequence $p_0, \ldots, p_t$ where $p_0 = p, p_t = q$ and $\forall i \ 0 \geq i \geq t-1 \quad (p_i, p_{i+1}) \in A$. The Boolean predicate $\text{path}_D(p, q)$ indicates the existence of a dipath from p to q in D.

We define now the set of vertices of D *reachable from* $p \in V$: $\text{Reach}_D(p) = \{q \in V \mid \text{path}_D(p, q) = 1\}$. And for all $U \subseteq V$ and $p \in V$, $AllReach_D(p, U) = \{D' = (V, A') \mid A' \subseteq A \wedge U \subseteq \text{Reach}_{D'}(p)\}$ is the set of all sub-graphs of D in which every vertices q in U are reachable from p.

We say that D is a *strongly connected graph* if there is a path between every pair of vertices. In other words $\forall p \in V \quad \text{Reach}_D(p) = V$. Note that if an undirected graph G is connected, $dir(G)$ is strongly connected.

Let $S \subseteq V$, we note $D_{|S}$ the graph induced from D by the vertices S. And we say that S is a *strongly connected component* (or SCC) if $D_{|S}$ is a maximal strongly connected subgraph of D.

2.2 Message Adversaries

In the general case, a message adversary is simply a set of infinite sequences of instant graphs. We only consider here "oblivious" or "iterated" message adversaries where instant graphs can be chosen in the same set which remains fixed all along the execution. In other words, the adversary does not choose by looking at the past. So given a fixed set of digraphs $\mathbf{M}$, we consider only the infinite sequences of elements of $\mathbf{M}$. Such a sequence is denoted $D^1, D^2, \cdots$ or $(D^i)_{i=1}^{\infty}$. The set of such infinite sequences is denoted $\mathbf{M}^\omega$. The set of finite sequences of length $r \in \mathbb{N}$ is denoted $\mathbf{M}^r$. Oblivious adversaries are simply the sets of digraphs $\mathbf{M}$.

Let $G = (V, E)$ be a graph and note $A = dir(E)$. We introduce two message adversaries. The f-omission message adversary $\mathbf{O}_f(G)$ is the set of all possible sub-graphs when at most f arcs can be removed from A:

▶ **Definition 1** (f-omission message adversary)**.**

$$\mathbf{O}_f(G) = \{D' = (V, A') \text{ digraph} \mid A' \subseteq A \wedge |A| - |A'| \leq f\} \ .$$

The f-half-duplex message adversary forbids the removal of two symmetric arcs between two vertices:

▶ **Definition 2** (f-half-duplex message adversary)**.**

$$\mathbf{HD}_f(G) = \{D' = (V, A') \mid D' \in \mathbf{O}_f(G) \wedge \forall p, q \in V \quad \{p, q\} \in E \wedge (p, q) \notin A' \Rightarrow (q, p) \in A'\} \, .$$

Note that by construction $\mathbf{HD}_f(G) \subseteq \mathbf{O}_f(G)$. Moreover, $\mathbf{HD}_f(G)$ contains all the tournament graphs with base G.

2.3 Execution of a Distributed Algorithm

A scenario is a sequence of instant graphs. We explain how to relate executions to scenarios. Given a oblivious message adversary $\mathbf{M}$, we define what is an execution of a given algorithm $\mathcal{A}$ with a given initial configuration ι. Every process can execute the following communication primitives:

- $send(msg)$ to send the same message msg to all out-neighbours,
- $recv()$ to get the messages from all in-neighbours.

An *execution*, or *run*, of algorithm $\mathcal{A}$ *subject to* scenario $\sigma \in \mathbf{M}^\omega$ is the following. Consider process u and one of its out-neighbours v in the underlying communication network. During round $r \in \mathbb{N}$, a message msg is sent from u to all its neighbours according to the instructions in algorithm $\mathcal{A}$. The node v will receive the corresponding message msg only if H, the r-th element of σ, is such that $(u, v) \in H$. All messages sent in a round can only be received in the same round. After sending and receiving messages, all processes update their states according to $\mathcal{A}$ and the messages they received. Given that all nodes have unique identities, when a message is received, it is known from which neighbour it is received. A *configuration* corresponds to the set of local states at the end of a given round.

Given $w \in \mathbf{M}^r$, and an initial configuration ι, let $s^p_\iota(w)$ denote the state of process p at the end of the r-th round of algorithm $\mathcal{A}$ subject to scenario w, with initial configuration ι. The initial state of p is therefore $\iota(p) = s^p_\iota(\varepsilon)$ where ε is the empty run. When ι is clear from the context, we might omit it and simply write $s^p(w)$. An *execution* of $\mathcal{A}$ subject to scenario $\sigma \in \rho(\mathbf{M})$ is the (possibly infinite) sequence of such message exchanges and corresponding configurations.

2.4 The k-Set Agreement Problem

We give the formal definition of the k-set agreement problem. This problem has been introduced in 1993 by [9].

Consider a system with n vertices. Given that each process has an initial value, each of them must decide a final value among the proposed values in such a way that there are at most k different decided values.

Validity: Any final value was the initial value of some process,
k-agreement: The set of final values is of size at most k,
Termination: Every process outputs a final value.
Note that when $n = k + 1$, this problem is also defined as the set agreement problem.

f \ k	1	2	3
1	yes	yes	yes
2	no	**yes**	yes
3	no	**no**	yes

(a) in $dir(K_3)$

f \ k	1	2	3
1	no	**yes**	yes
2	no	**no**	yes
3	no	no	yes

(b) in $dir(P_3)$

Figure 1 Solvability of the k-set agreement according to the number of omission faults f in $dir(K_3)$ and $dir(P_3)$.

3 2-Set Agreement in the K_3 and P_3 Topology

In this section we characterize the solvability of the 2-set agreement problem according to the number of possible omission faults f in graphs $dir(K_3)$ and $dir(P_3)$. We note $\Pi = \{p_1, p_2, p_3\} = \{\bullet, \bullet, \circ\}$ the set of processes in the network.

Figure 1 depicts both $dir(K_3)$ and $dir(P_3)$ and the solvability of the k-set agreement with respect to the number of omission faults f when $k \leq 3$ and $f \leq 3$. The $k = 3$ case is trivially solvable but highlights the border between the solvability and unsolvability of the k-set agreement. We prove below the four results highlighted in bold font in the table. Other results of the table are either straightforward or already well known, see e.g. [10]. This table completely characterizes the 2-set agreement for K_3 and P_3.

▶ **Proposition 3.** *The 2-set agreement problem is solvable in $\mathbf{HD}_f(K_3)$ if and only if $f \leq 2$.*

Proof. For the necessary part, we show that the 2-set agreement is impossible in $\mathbf{HD}_3(K_3)$. To this end, we extract from $\mathbf{HD}_3(K_3)$ a subset $\mathcal{S}$ of digraphs (its poset by arc inclusion is shown in Figure 2) such that we can construct a subdivision of the triangle by gluing graphs together by identifying the views of the processes (the in-neighbourhood of the vertices). More formally, let two digraphs $D_1 = (\Pi, A_1)$ and $D_2 = (\Pi, A_2)$ of $\mathcal{S}$. D_1 and D_2 are glued together by identifying p_i and p_j of Π if $N^-_{D_1}(p_i) = N^-_{D_2}(p_i)$ and $N^-_{D_1}(p_j) = N^-_{D_2}(p_j)$. The newly created object after identifying all such views is depicted in Figure 3 and corresponds indeed to a subdivision of the triangle.

Suppose there exists a 2-set agreement algorithm. From a distributed system point of view, the triangles which form the subdivision correspond to all possible 1-round execution. And we can iterate the subdivision to represent all possible executions of a certain round. Now, if we look closer to the subdivision and the represented digraphs, we see that the processes standing in the corner of the subdivided triangle doesn't receive any messages, they thus decide their own value. Moreover, those on the edges receive messages from processes in the corresponding corners, and those inside receive messages from every processes.

Consider the values decided by a all executions of the 2-set agreement algorithm. What we get here is a Sperner colouring where the colour of a vertex in the subdivision is the decision value of the associated process: every algorithm run on $\mathcal{S} \subseteq \mathbf{HD}_3(K_3)$ that terminates leads to a decision of each process as described, and thus to a Sperner colouring on the iterated subdivided figure. Now, by Sperner's Lemma (see e.g. [19]) we have that for any Sperner colouring on a subdivided triangle, there exists at least one 2-dimensional simplex coloured

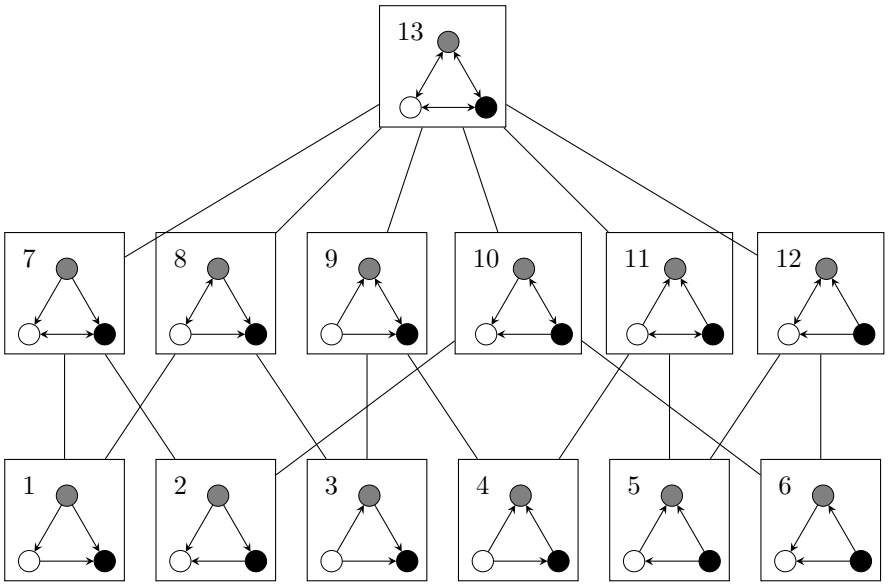


Figure 2 Poset of $\mathcal{S}$.

by three different colours, i.e. an execution where the three processes decide three different values. The 2-agreement condition of the problem is thus violated.

For the sufficient part, we give a simple flooding algorithm that solves the 2-set agreement in a single round in K_3 with 2 or less omission faults: we fix a priority order (which can be arbitrary) that is known by all processes $p \in \Pi$. Now, each process decides the value of its candidate after one round (i.e. the process with the highest priority that it is aware of).

To prove the correctness of this algorithm, fix the priority order π: $\forall p_i \in \Pi \quad \pi(p_i) = i$. For the algorithm to work, all it takes is that a message is received by a process having a lower priority than the sender: if A is the set of arcs of the digraph in the first round, it is sufficient that there exists $(p_i, p_j) \in A$ such that $\pi(p_i) > \pi(p_j)$ because in that case p_j will have p_i as candidate and so p_i and p_j will have the same candidate after one round, thus deciding the same value. In $dir(K_3)$, there are 3 such arcs (whatever π), and as $f \leq 2$, there is at least one in every $D \in \mathbf{HD}_2(K_3)$. ◀

▶ **Proposition 4.** *The 2-set agreement problem is solvable in $\boldsymbol{HD}_f(P_3)$ if and only if $f \leq 1$.*

Proof. As for $dir(K_3)$, for the necessary part, we extract from $\mathbf{HD}_2(dir(P_3))$ a subset of instant digraphs that forms a subdivision of the triangle (Figure 4). This subset actually consist of all graphs of $\mathbf{HD}_2(dir(P_3))$.

The proof argument is the same as for Proposition 3. Yet, we notice that the vertices $\circ$ and $\bullet$ in the inside doesn't receive the messages from all other vertices ; and three inside triangles (the topmost ones) correspond to the same instant digraph. This is not a problem because as the states of the processes constituting these triangles are the same, they will be coloured the same way, i.e. the association between an execution and a triangle in the graph still stands. Moreover, the processes colours (decision value) satisfies a Sperner colouration: nodes on an edge are coloured by one of the colours of its end points and nodes in the interior are coloured by one of the colours of the triangle. Thus, we can apply Sperner's lemma as in the classical proof (see [13, Chap. 9]).

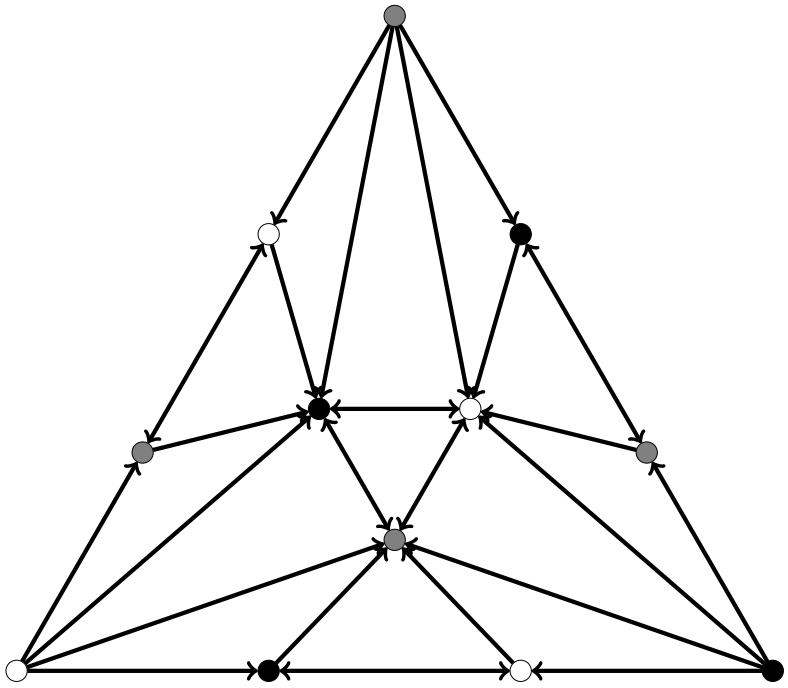


Figure 3 Subdivision constructed from $\mathcal{S} \subseteq \mathbf{HD}_3(\mathcal{K}_3)$.

The necessary part uses the same algorithm as in the proof of Proposition 3: in $dir(P_3)$ there are 2 arcs (p_i, p_j) with $\pi(p_i) > \pi(p_j)$, and $f \leq 1$ so there is at least one in each $D \in \mathbf{HD}_1(P_3)$. ◀

4 Solvability of 2-Set Agreement for Arbitrary Graphs

Before stating the characterization for graphs of arbitrary size, let's introduce first some notations and an important lemma.

4.1 Notations and the Causal Influence Lemma

We generalize the standard notion of *edge-connectivity* by introducing a new parameter ℓ allowing us to define the number of connected components.

▶ **Definition 5.** An undirected graph $G = (V, E)$ is (k, ℓ)-*edge-connected* ($\ell > 0$) if and only if $\forall E' \subseteq E \quad |E'| \leq k \Rightarrow G' = (V, E \setminus E')$ have at most ℓ connected components.

In other words, G cannot be cut into $\ell + 1$ connected components with only k edges removed.

▶ **Definition 6.** The ℓ-edge-connectivity of a graph G, denoted by $c_\ell(G)$, is the largest possible k such that G is (k, ℓ)-edge-connected.

In particular, the classical edge-connectivity $conn(G)$ corresponds to $c_1(G) + 1$: it is the minimal number of edges we need to remove to be able to disconnect the graph in two connected components.

For $dir(G) = (V, A)$, we remark that all sub-digraphs $dir(G') = (V, A')$ such that $|A| - |A'| \leq c_\ell(G)$ have at most ℓ strongly connected components.

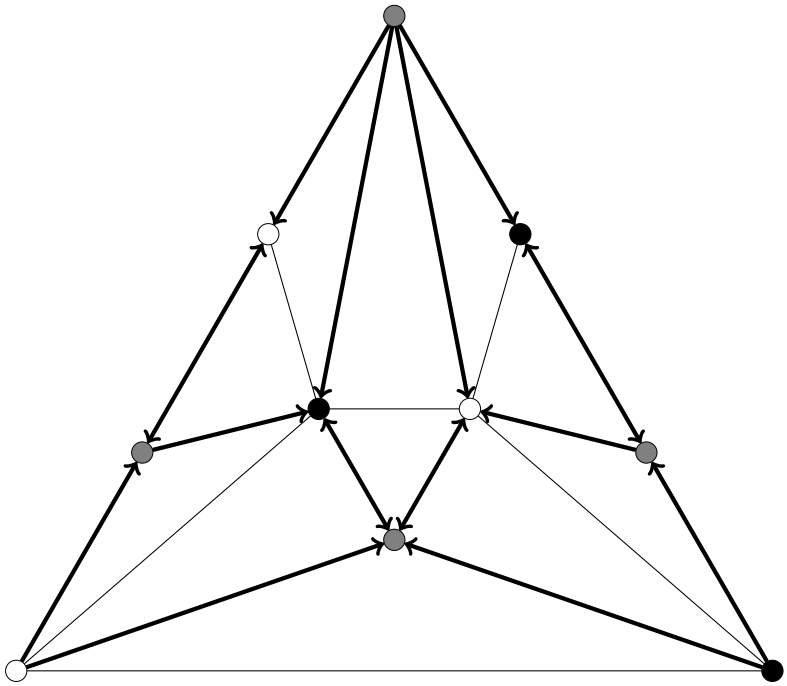


Figure 4 Subdivision constructed from $\mathbf{HD}_2(P_3)$.

Indeed, assume the opposite, this means that there exists $c_\ell(G)$ arcs that can be removed in $dir(G)$ to form more than ℓ strongly connected components in $dir(G)$. Now, remove the corresponding set of edges in G, there are no more than $c_\ell(G)$ of them, and one would get more than ℓ connected components in G. A contradiction with the definition of $c_\ell(G)$.

Given $p \in \Pi$, we note x_p the initial value of process p, and $\text{Info}_p(r)$ the set of initial values known by p at round r. Let $N_p^r = N_{D^r}^-(p)$ be the set of source processes of incoming arcs of p at round r. In a full-information protocol, Info_p is defined as:

$$\begin{aligned} &\text{Info}_p(0) = \{x_p\}\,, \\ \forall r > 0 \quad &\text{Info}_p(r) = \text{Info}_p(r-1) \cup \{x_q \mid q \in N_p^r\}\,. \end{aligned}$$

Remark that Info_p can only grow or stay fixed from round to round, i.e. the processes do not forget the values they have seen in the past.

Denote $\mathcal{I}_{p,b}^a$ the set of processes that know at round a at least the information that p had at round b. Formally:

$$\mathcal{I}_{p,b}^a = \{q \in \Pi \mid \text{Info}_p(b) \subseteq \text{Info}_q(a)\}\,.$$

If p or b can be deduced from the context, we will simply write $\mathcal{I}_p^a$ or even $\mathcal{I}^a$.

Recall that $AllReach_D(p, S)$ is the set of all sub-graphs of D in which every vertices $q \in S$ are reachable from p.

The following lemma expresses the fact that after $n-1$ occurrences of digraphs in which some processes are reachable from $p \in V$, they will have the information that p had.

▶ **Lemma 7** (Causal Influence Lemma). *Let $\sigma = (D^i)_{i=1}^\infty$ be a sequence of instant digraphs and $t \geq n-1$. Let $S \subseteq V$; if there exists an increasing sequence of indices $1 \leq i_1 < \ldots < i_t$ such that $\forall 1 \leq j \leq t \quad D^{i_j} \in AllReach_D(p, S)$, then in a full information protocol, we have:*

$$\forall q \in S \quad \forall i_{t'} > i_t \quad \text{Info}_p(i_1) \subseteq \text{Info}_q(i_{t'})\,.$$

Proof. Note $\forall r \geq 0 \quad \mathcal{I}^r = \mathcal{I}^r_{p,i_1}$. Consider the worst case where $\mathcal{I}^{i_1} = \{p\}$. Let $1 < a \leq t$; if $\mathcal{I}^{i_a} \not\subseteq S$, then because $D^{i_a} \in \mathit{AllReach}_D(p, S)$, we necessarily have an arc from a vertex $q \in \mathcal{I}^{i_a}$ to $q' \in V \setminus \mathcal{I}^{i_a}$, and so $|\mathcal{I}^{i_a+1}| \geq |\mathcal{I}^{i_a}| + 1$, i.e. at each occurrence of a digraph of $\mathit{AllReach}_D(p, S)$, the number of processes informed by $\mathrm{Info}_p(i_1)$ increases by at least 1. Thus, as $|S| \leq n$, $t \geq n-1$ and in the worst case $|\mathcal{I}^{i_1}| = 1$, we have $S \subseteq \mathcal{I}^{i_{t'}}$ for all $t' > t$. ◀

Let π be a priority order over Π, we define $\forall p \in \Pi \quad \forall r \geq 0 \quad \mathrm{cand}^\pi_p(r) \in \mathrm{Info}_p(r)$ the *candidate* of $p \in \Pi$ at round r as the process with the highest priority known by p, with respect to π:

$$\mathrm{cand}^\pi_p(r) = \underset{q \in \mathrm{Info}_p(r)}{\arg\max}\, \pi(q)\,.$$

When π can be easily deduced from the context, we will sloppily write $\mathrm{cand}_p(r)$.

4.2 2-Set Agreement Characterization for Arbitrary Graphs

▶ **Theorem 8.** *$\forall G = (V, E)$ such that $|V| = n \geq 3$, the 2-set agreement problem is solvable despite f omission faults if and only if $f \leq c_2(G)$.*

Proof.

Sufficient part

The algorithm for the sufficient part acts as a full information protocol and processes decide after a sufficiently long time T (to be bounded later). It is based, once again, on a priority order π over V and it solves the 2-set agreement in $\mathbf{HD}_f(G)$ for all G if $f \leq c_2(G)$.

Denote $p^* \in V$ the process with the highest priority: $p^* = \underset{p \in V}{\arg\max}\, \pi(p)$ and note $\mathcal{I}^r = \mathcal{I}^r_{p,0}$ the set of processes "informed" by the value of p^* at round r and $\overline{\mathcal{I}^r} = V \setminus \mathcal{I}^r$ its complement. By definition, we have that $\forall r \geq 0 \quad \forall p \in \mathcal{I}^r \ \mathrm{cand}_p(r) = x_{p^*}$. We define the stability property of $\mathcal{I}$:

$$\forall a \leq b \quad \mathrm{stable}^b_a = \begin{cases} 1 & \text{if } \ \forall a \leq r, r' \leq b \quad \mathcal{I}^r = \mathcal{I}^{r'} \\ 0 & \text{otherwise} \end{cases}$$

The following lemma expresses the fact that after a stability period of n rounds, there are at most two candidates among all processes.

▶ **Lemma 9.** *Applying a full-information protocol, if there exist $a, b \geq 0$ such that $b - a \geq n$ and if $\mathrm{stable}^b_a = 1$ then*

$$\forall r \geq b \quad \left| \bigcup_{p \in V} \mathrm{cand}_p(r) \right| \leq 2\,.$$

Proof. $\forall a \leq b$, a stability period between a and b (i.e. $\mathrm{stable}^b_a = 1$) necessarily implies that we are in one of the following two configurations:

1. Either $\mathcal{I}^a$ is "saturated", i.e. $\mathcal{I}^a = V$, and $\forall r \geq a \quad \mathcal{I}^r = V$
2. Or there are no communication from $\mathcal{I}^r$ to $\overline{\mathcal{I}^r}$: $\forall (p, q) \in A$ and $\forall a \leq r \leq b$, $p \in \mathcal{I}^r$ implies that $q \in \mathcal{I}^r$. In other words, there are no messages from any process $p \in \mathcal{I}^r$ to a process $q \in \overline{\mathcal{I}^r}$. Indeed, the contrary would imply that q receive x_{p^*} and thus would be in $\mathcal{I}^r$.

In the first case, by definition of $\mathcal{I}$, we have that $x_{p^*} \in \text{Info}_p(b)$ for all processes $p \in V$, thus the only candidate after round b is x_{p^*}.

Now for the second case, consider the sequence $(D^1, D^2, \ldots)$ of instant digraphs subgraphs of $dir(G)$. We have that in every round $a \leq r \leq b$ the D^r has more than one strongly connected component. And in fact, it has exactly two SCC because $f \leq c_2(G)$ which by definition means that D^r has at most 2 SCC, which can only be $\mathcal{I}^r$ and $\overline{\mathcal{I}^r}$. In other words, the sub-graph $D^r_{|\overline{\mathcal{I}^r}}$ is a strongly connected graphs, so $\forall p \in \overline{\mathcal{I}^r}$, $D^r \in AllReach_{dir(G)_{|\overline{\mathcal{I}^r}}}(p, \overline{\mathcal{I}^r})$. In particular, consider $p^+ = \arg\max_{p \in \overline{\mathcal{I}^r}} \text{cand}_p(a)$ the process that has the best candidate over all $\overline{\mathcal{I}^r}$ at round a. Now, applying Lemma 7 with $i_1 = a, i_2 = a+1, \ldots, i_t = b-1$ and $S = \overline{\mathcal{I}^r}$ gives us that $\forall q \in \overline{\mathcal{I}^r} \quad \forall r \leq b \quad \text{Info}_{p^+}(a) \subseteq \text{Info}_q(r)$, i.e. by round b, all processes of $\overline{\mathcal{I}^b}$ have $\text{cand}_{p^+}(a)$ as candidate. Now, $\text{cand}_{p^+}(a)$ combined with x_{p^*} known by all processes of $\mathcal{I}^r$, leads to at most two different candidates for all processes at round b.

And this holds for all the following rounds $r \geq b$ because $\forall p \in \overline{\mathcal{I}^b}$, a change of candidate at round r $\text{cand}_p(r) \neq \text{cand}_p(b)$ necessarily means that $\text{cand}_p(r) = x_{p^*}$: a message to from a process $q \in \mathcal{I}^{r-1}$ was received by p. ◀

Such a period of stability of n rounds must occurs before round n^2, or all processes are in $\mathcal{I}^{n^2}$: A "non stable" period from a to b of n rounds ($\text{stable}_a^b = 0$ with $b - a = n$) necessarily implies that $\mathcal{I}^b \geq \mathcal{I}^a + 1$, i.e. at least one process get informed by x_{p^*} during this period. So if n consecutive such periods occur then all processes are informed; otherwise a period of n rounds has occurred and Lemma 9 tells us there are at most two different candidates. Thus, all processes $p \in V$ deciding value $x_{\text{cand}_p(n^2)}$ respect the agreement property of the 2-set agreement and the problem is solved.

Necessary part

For the impossibility part, we proceed by reduction to $dir(P_3)$ and $dir(K_3)$: we suppose there exists an algorithm $\mathcal{A}$ that solves the 2-set agreement despite $f = c_2(G) + 1$ omission faults, and we construct an adversary for $dir(G)$ corresponding to an adversary for $dir(K_3)$ and $dir(P_3)$ for which the 2-set agreement is impossible to solve.

Let's first consider a decomposition of the vertices of G in 3 sets: $V = V_1 \cup V_2 \cup V_3$ corresponding to the definition of $c_2(G)$. Denote $d_{i,j}$ the number of edges between V_i and V_j in G; and $A_{i,j}$ the set of arcs in $dir(G)$ from V_i to V_j. By definition, we have that $|A_{i,j}| = |A_{j,i}| = d_{i,j} = d_{j,i}$. Let $H_i = (V_i, A_{i,i}) = G_{|V_i}$ $(1 \leq i \leq 3)$ the associated components in $dir(G)$. The decomposition is chosen such that $d_{1,2} + d_{2,3} + d_{3,1} = c_2(G) + 1 = f$. Consequently, we have:

$$\left(\bigcup_{1 \leq i \leq 3} V_i, \bigcup_{1 \leq i,j \leq 3} A_{i,j} \right) = dir(G)$$

Suppose the problem is solvable for $f = c_2(G) + 1$, i.e. there exists an algorithm $\mathcal{A}$ such that for each sequence of instant digraphs of $\mathbf{O}_{c_2(G)+1}(G)$, $\mathcal{A}$ terminates and solves the 2-set agreement problem. In particular, consider the message adversary $\mathbf{D}$ that removes only one given set $A_{i,j}$ of arcs between two components H_i and H_j $(i \neq j)$, three times.

Formally:

$$\mathbf{D} = \{(V, dir(E) \setminus \{A_{i_1,j_1} \cup A_{i_2,j_2} \cup A_{i_3,j_3}\}) \mid \\ \forall 1 \leq a, b \leq 3 \quad i_a \neq j_a \text{ and } a \neq b \Rightarrow ((i_a, j_a) \neq (i_b, j_b) \wedge (i_a, j_a) \neq (j_b, i_b))\} .$$

We indeed have the inclusion $\mathbf{D} \subseteq \mathbf{O}_f(G)$ because the f-half-duplex condition necessarily implies that $|A_{i_1,j_1} \cup A_{i_2,j_2} \cup A_{i_3,j_3}| \leq f$. Note that if the decomposition in H_i of G forms a chain and not a clique, then one of the A_{i_a,j_a} is empty.

We now show that solving with $\mathcal{A}$ for $\mathbf{D}$ implies a solution for $\mathbf{HD}_3(\Gamma)$ where Γ has three processes: $\Pi = \{p_1, p_2, p_3\} = \{\circ, \bullet, \bullet\}$, with either $\Gamma = K_3$, or $\Gamma = P_3$ depending on the decomposition of G. To do so, we define a new algorithm $\mathcal{A}_\Gamma$ that simulates $\mathcal{A}$ in Γ such that every sequence $\sigma \in \mathbf{D}$ corresponds to a sequence $\sigma' \in \mathbf{HD}_3(\Gamma)$ where p_i simulates the processes V_i of H_i.

For every variable var used in $\mathcal{A}$, we note var_u^r the value of var for process $u \in V$ at round r. For each of these variables, the processes $p_i \in \Pi$ hold an array of values $\overline{var}_{p_i}^r$ for every round r where $\forall u \in V \quad \overline{var}_{p_i}^r[u]$ corresponds to the simulated value var_u^r for process u. For all $u, v \in V$, we note $m^r(u, v)$ the message possibly sent from u to v at round r by $\mathcal{A}$. Every process $p_i \in \Pi$ simulates the execution of $\mathcal{A}$ by V_i as such:

At round r, $p_i \in \Pi$ performs the two following steps:

1. Compute and locally record the values $\overline{var}_{p_i}^r[u]$ for all processes $u \in V_i$ by simulating their local behaviour and the content of messages $m^r(u, v)$ for all $u, v \in V_i$.
2. Send to process $p_j \in \Pi$ the concatenation of messages $m^r(u, v)$ where $u \in V_i$ and $v \in V_j$.

The step 1 corresponds to local computations in V_i and to the sending by u and recieving by v (update of v's state) of the messages transiting on $A_{i,j}$ by algorithm $\mathcal{A}$. Step 2 corresponds to messages transiting on $A_{i,j}$ by $\mathcal{A}$. We thus have the equality $\overline{var}_{p_i}^r[u] = var_u^r$ that stands for all $r \geq 0$ and all $u \in V_i$.

By our assumption on $\mathcal{A}$, all processes of V decide an output value that satisfies the 2-agreement property required. When the termination occurs, $p_i \in \Pi$ chooses the smallest value among the ones of its simulated processes V_i. This guarantees that the total number of different decided values in Γ isn't larger than those decided with $\mathcal{A}$. So this solves the 2-set agreement problem on Γ in $\mathbf{HD}_3(\Gamma)$, which is a contradiction with Proposition 3 (K_3) or 4 (P_3). ◀

5 General Case

5.1 Impossibility Proof for Graphs of Size $k + 1$

In this section, we assume $|G| = k + 1$. Let Δ^k be the standard k-dimensional simplex and let $G = (V, E)$ be a communication graph with $k + 1$ vertices. The vertices of G and Δ^k are indexed by the elements of $[k] = \{0, \ldots, k\}$. Using the description of the standard chromatic subdivision used in [16], we define a subdivision of the simplex Δ^k corresponding to G, which we denote $Chr(G)$.

For this purpose, we will use the author's notation, for a full description, see [16]. Our construction of $Chr(G)$ is in many points similar to the construction of [16], $\Sigma_{\mathcal{P}}(K)$ with $K = \Delta^k$ and $\mathcal{P}$ being the infinite family of cross-polytopes. The difference lies in subdivided simplices considered at each step: at step j, instead of replacing each $k - j + 1$-dimensional simplices of K (denoted $K^{(k-j+1)}$) by the corresponding Schlegel diagram, we only replace those in which their corresponding sub-graph has at least one edge. More formally we simply replace the definition of $K^{(i)}$ as such:

$$K_G^{(i)} = \{\sigma \in K \mid |\sigma| = i \wedge \mathbb{P}_2(\sigma) \cap E \neq \emptyset\}$$

where $\mathbb{P}_2(\sigma) = \{\tau \subseteq \sigma \mid |\tau| = 2\}$ is the set of all possible edges between the vertices of σ. Now, using the author's inductive steps we construct $X_1, \ldots, X_k$ and set $Chr(G) = X_k$.

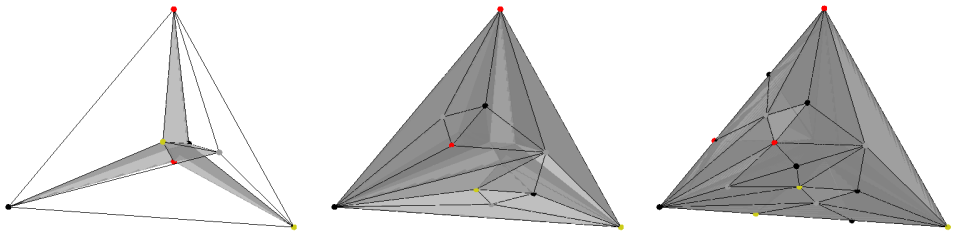


Figure 5 Construction steps of $Chr(S_3)$.

To visualize how $Chr(G)$ is constructed, we give an example for a graph with 4 vertices: the star S_3 (with $\bullet$ at the center). The subdivision is constructed in three steps ($\mathcal{P}_d$ is the Schlegel diagram of the cross-polytope of dimension d):

1. replace Δ^3 with $\mathcal{P}_3$;
2. for each faces σ of dimension 2 such that its corresponding sub-graph has at least one edge, replace σ by $\mathcal{P}_2$ and join the result with the previous step, thus subdividing a top-dimensional simplex;
3. same as 2 with faces of dimension 1, namely edges.

This construction is presented in Figure 5. For visualization comfort, we didn't grayed all 3-dimensional simplices for the first step.

The Figure 4 in Section 3 is another example, it represents $Chr(P_3)$.

▶ **Proposition 10.** *$Chr(G)$ is a simplicial subdivision of Δ^k.*

Proof. Comparing to [16], we only prevented the subdivision of some $k-i+1$-simplices of Δ^k at step i. Yet, in the construction of [16], the transition of X_{i-1} to X_i preserves the fact that the resulting complex is a subdivision. Thus, by *not* subdividing simplices, the resulting complex is still a subdivision of Δ^k. ◀

Now, for each k-dimensional simplex σ of $Chr(G)$, we associate a directed graph γ_σ. To define it, recall the combinatorial description of the k-simplices in X_ℓ presented in [16]: they are in the form $\sigma = ((i_1, A_1), \ldots, (i_{k+1}, A_{k+1}))$ with $\{i_1, \ldots, i_{k+1}\} = [k]$ and satisfies some conditions on the A_is. Our construction only changes this description by adding a condition on the intersection of the edges of G and the A_is, preventing the creation of undesirable k-simplices.

We define corresponding graphs as follows:

$$\forall \sigma \in Chr(G) \quad \gamma_\sigma = ([k], \{(i_a, i_b) \mid \{i_a, i_b\} \in E \wedge A_a \subseteq A_b\}) .$$

And let $\Gamma(G) = \{\gamma_\sigma \mid \sigma \in Chr(G)\}$ be the set of all such digraphs.

▶ **Proposition 11.** *For all communication graph $G = (V, E)$ with $|E| = m$, we have*

$$\Gamma(G) \subseteq \boldsymbol{HD}_m(G) .$$

Proof. Every γ_σ has the same set of vertices than G, namely $[k]$ and for each edge $\{i_a, i_b\} \in E$, we add one or two arcs depending on the inclusion order of A_a and A_b. Thus γ_σ is a sub-digraph of G and has at least m arcs, so it indeed lies in $\mathbf{HD}_m(G)$. ◀

Before stating the main lemma of this section, we need to remark a useful fact about ℓ-edge-connectivity.

▶ **Lemma 12.** *For any communication graph $G = (V, E)$ with $|V| = k + 1$ and $|E| = m$, we have*

$$c_k(G) = m - 1 \,.$$

Proof. In order to have $|V| = k + 1$ connected components, we need to remove every edge. Now, removing $m - 1$ edges guaranties us to have exactly one connected components with two vertices, thus we have no more than k connected components. ◀

▶ **Lemma 13.** *For any communication graph $G = (V, E)$ with $|V| = k + 1$ and $|E| = m$, the k-set agreement problem is not solvable if $c_k(G) + 1$ or more omission faults can occur.*

Proof. First, note following what we just proved above, $c_k(G) + 1 = m$. Now, we roughly use the same technique we used in Section 3 to prove the impossibility of the 2-set agreement in the K_3 and P_3 topology. The difference is that instead of extracting a subset of $\mathbf{HD}_m(G)$ that can form a subdivision of Δ^k, we rather construct the subdivision $Chr(G)$ which is indeed a subdivision of Δ^k. Sperner's lemma tells us that at least one k-dimensional simplex has its vertices coloured with $k + 1$ different colours. Thus k-set agreement is not solvable for the adversary consisting of the graphs corresponding to the k-dimensional simplices of $Chr(G)$ – namely $\Gamma(G)$. And Proposition 11 yields that $\Gamma(G) \subseteq \mathbf{HD}_m(G)$ and by definition $\mathbf{HD}_m(G) \subseteq \mathbf{O}_m(G)$; thus the result follows. ◀

5.2 General Proof

We can now state the main theorem of the paper which fully characterize the solvability of the k-set agreement problem in the model of the paper.

▶ **Theorem 14.** *Let $k \in \mathbb{N}$ and $G = (V, E)$ be any communication network. The k-set agreement problem is solvable despite f omission faults if and only if $f \leq c_k(G)$.*

Proof. To show the impossibility part, i.e. the k-set agreement is not solvable if $f \geq c_k(G)+1$, we reduce G to the case of a smaller graph that has $k + 1$ vertices and we use Lemma 13 to prove the impossibility.

Let $\mathcal{G}(n)$ be the set of all undirected graphs with n vertices.

Let's first consider a partition of the vertices of G in $k + 1$ non-empty sets: $V = V_1 \cup \ldots \cup V_{k+1}$ associated with $c_k(G)$. Denote $d_{i,j}$ the number of edges between V_i and V_j in G; and $A_{i,j}$ the set of arcs in $dir(G)$ from V_i to V_j (some may empty):

$$A_{i,j} = \{(u, v) \mid u \in V_i \wedge v \in V_j\} \,.$$

By definition, we have that $|A_{i,j}| = |A_{j,i}| = d_{i,j} = d_{j,i}$. The partition is chosen such that $\sum_{1 \leq i,j \leq k+1} d_{i,j} = c_k(G) + 1 = f$.

Let G' be the communication graph obtained by contracting each of the $k + 1$ sets V_i into a single process p_i (and removing redundant edges). Denote Π the set of G''s processes and m its number of edges. Remark that $|\{\{i, j\} \mid 1 \leq i, j \leq k + 1 \text{ and } d_{i,j} \neq 0\}| = m$.

Suppose the problem solvable for $f = c_k(G) + 1$, i.e. there exists an algorithm $\mathcal{A}$ such that for each sequence of instant digraphs of $\mathbf{O}_{c_k(G)+1}(G)$, $\mathcal{A}$ terminates and solves the k-set agreement. In particular, consider the message adversary $\mathbf{D}$ that removes – at most m times – only one given set $A_{i,j}$ of arcs between two components V_i and V_j $(i \neq j)$. Formally, we

define $\mathcal{F}$ as the family of sets of arcs that satisfies this condition:

$$\mathcal{F} = \left\{ \bigcup_{1 \leq a \leq m} A_{i_a,j_a} \mid 1 \leq i_a, j_a \leq k+1 \text{ and } i_a \neq j_a \text{ and } \forall 1 \leq b \leq m \quad a \neq b \Rightarrow (i_a, j_a) \neq (j_b, i_b) \right\} .$$

Now $\mathbf{D}$ is defined as

$$\mathbf{D} = dir(G) \cup \{(V, dir(E) \setminus F) \mid F \in \mathcal{F}\} .$$

We indeed have the inclusion $\mathbf{D} \subseteq \mathbf{O}_f(G)$ because of the choice of the V_is and the half-duplex condition, we necessarily have for each $F \in \mathcal{F}$, $|F| \leq f$.

We now show that solving the k-set agreement problem with $\mathcal{A}$ for $\mathbf{D}$ implies a solution for $\mathbf{HD}_m(G')$. To do so, we define a new algorithm $\mathcal{A}_{G'}$ that simulates $\mathcal{A}$ in G' such that every sequence $\sigma \in \mathbf{D}$ corresponds to a sequence $\sigma' \in \mathbf{HD}_m(G')$ where p_i simulates the processes V_i.

For every variable var used in $\mathcal{A}$, we note var_u^r the value of var for every process $u \in V$ at round r. In $\mathcal{A}_{G'}$, each process $p_i \in \Pi$ holds an array of values $\overline{var}_{p_i}^r$ for every round r where $\forall u \in V_i \quad \overline{var}_{p_i}^r[u]$ corresponds to the simulated value var_u^r for process u. For all $u, v \in V$, we note $m^r(u, v)$ the message possibly sent from u to v at round r by $\mathcal{A}$. Every process $p_i \in \Pi$ simulates the execution of $\mathcal{A}$ by V_i as such:

At round r, $p_i \in \Pi$ performs the two following steps:

1. According to $\mathcal{A}$, compute and locally record the values $\overline{var}_{p_i}^r[u]$ for all processes $u \in V_i$ by simulating their local behaviour and the content of messages $m^r(u, v)$ for all $u, v \in V_i$.
2. Send to process $p_j \in \Pi$ the concatenation of messages $m^r(u, v)$ where $u \in V_i$ and $v \in V_j$.

The step 1 corresponds to local computations in V_i and to the expedition and reception of the messages transiting on $A_{i,i}$ by algorithm $\mathcal{A}$. Step 2 corresponds to messages transiting on $A_{i,j}$ by $\mathcal{A}$. We thus have the equality $\overline{var}_{p_i}^r[u] = var_u^r$ that stands for all $r \geq 0$ and all $u \in V_i$.

By our assumption on $\mathcal{A}$, all processes of V decide an output value that satisfies the agreement condition of the k-set agreement problem. When this termination occurs, in $\mathcal{A}_{G'}$ $p_i \in \Pi$ chooses the smallest value among the ones of its simulated processes V_i. This guarantees that the total number of different decided values in $\mathcal{A}_{G'}$ isn't larger than those decided with $\mathcal{A}$. So this solves k-set agreement problemin $\mathbf{HD}_m(G')$, which is a contradiction with Lemma 13.

For the sufficient part, the k-set agreement algorithm is the exact extension of what is presented in Section 4.2. ◀

6 Conclusion

In this note, we give a complete characterization of the number of omission faults a communication network with arbitrary topology can withstand when solving the k-set agreement problem for any given $k \in \mathbb{N}$. Introducing a new combinatorial parameter c_k that is an extension of the classical edge-connectivity of graphs, we have shown that, given a graph G, the k-set agreement problem can be solved if and only if the number of omission faults per round is less than $c_k(G)$. The general solvability of the k-set agreement problem in oblivious message adversaries is still open but an important sub-class has been solved in this paper.

Note that we proved the necessary condition using a new twist for topological methods in distributed computability: we allow some one-step events to correspond to more than one simplex in the round diagram. This new technique is of independent interest and could open new impossibility proofs for other distributed problems. We recently found a way to prove (part of) the same result using a reduction from the well-known complete topology case. However, the relationship between this new topological technique and general reduction techniques is yet to investigate.

Acknowledgement. The authors wish to thanks Damien Imbs for fruitful discussions on the 2-set agreement case.

References

1. Yehuda Afek and Eli Gafni. *Asynchrony from Synchrony*, pages 225–239. Number 7730 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.
2. Martin Biely, Peter Robinson, and Ulrich Schmid. Easy impossibility proofs for k-set agreement in message passing systems. In *OPODIS*, volume 7109 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2011.
3. Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Networked Systems – Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, volume 9466 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2015.
4. Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks, 2016. submitted.
5. E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing*, 1993.
6. Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *STOC'93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 91–100, New York, NY, USA, 1993. ACM Press. doi:10.1145/167088.167119.
7. Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'97, pages 189–198. ACM, 1997. doi:10.1145/259380.259439.
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. doi:10.1007/s00446-009-0084-6.
9. S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, Jul 1993.
10. Étienne Coulouma, Emmanuel Godard, and Joseph G. Peters. A characterization of oblivious message adversaries for which consensus is solvable. *Theor. Comput. Sci.*, 584:80–90, 2015. doi:10.1016/j.tcs.2015.01.024.
11. Eli Gafni and Petr Kuznetsov. The weakest failure detector for solving k-set agreement. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC'09, pages 83–91, New York, NY, USA, 2009. ACM. doi:10.1145/1582716.1582735.
12. Rachid Guerraoui, Petr Kouznetsov, and Bastian Pochon. A note on set agreement with omission failures. *Electr. Notes Theor. Comput. Sci.*, 81, 2003.

13 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 1 edition edition, 2013.

14 Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. Computability in distributed computing: A tutorial. *SIGACT News*, 43(3):88–110, 2012.

15 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

16 Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology Homotopy Appl.*, 14(2):197–209, 2012. URL: `http://projecteuclid.org/euclid.hha/1355321488`.

17 Michel Raynal. *Set Agreement*, pages 1956–1959. Springer New York, New York, NY, 2016. `doi:10.1007/978-1-4939-2864-4_367`.

18 Michel Raynal and Julien Stainer. Synchrony weakened by message adversaries vs asynchrony restricted by failure detectors. In Panagiota Fatourou and Gadi Taubenfeld, editors, *PODC*, pages 166–175. ACM, 2013.

19 M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. on Computing*, 29:1449–1483, 2000.

20 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS*, pages 304–313, 1989.

21 Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theor. Comput. Sci.*, 384(2-3):232–249, 2007.

Using Read-k Inequalities to Analyze a Distributed MIS Algorithm*†

Sriram Pemmaraju[1] and Talal Riaz[2]

1 **Department of Computer Science, University of Iowa, Iowa City, USA**
`sriram-pemmaraju@uiowa.edu`
2 **Department of Computer Science, University of Iowa, Iowa City, USA**
`talal-riaz@uiowa.edu`

Abstract

Until recently, the fastest distributed MIS algorithm, even for simple graph classes such as unoriented trees that can contain large independent sets within neighborhoods, has been the simple randomized algorithm discovered independently by several researchers in the late 80s. This algorithm (commonly called *Luby's algorithm*) computes an MIS of an n-node graph in $O(\log n)$ communication rounds (with high probability). This situation changed when Lenzen and Wattenhofer (PODC 2011) presented a distributed (randomized) MIS algorithm for unoriented trees running in $O(\sqrt{\log n} \cdot \log\log n)$ rounds. This algorithm was slightly improved by Barenboim et al. (FOCS 2012), resulting in an $O(\sqrt{\log n \cdot \log\log n})$-round (randomized) MIS algorithm for trees. At their core, these algorithms still run Luby's algorithm, but only up to the point at which the graph has been "shattered" into small connected components that can be independently processed in parallel.

The analyses of these tree MIS algorithms critically depends on "near independence" among probabilistic events, a feature that arises from the tree structure of the network. In their paper, Lenzen and Wattenhofer express hope that their algorithm and analysis could be extended to graphs with bounded arboricity. We show how to do this in the current paper. By using a new tail inequality for *read-k families* of random variables due to Gavinsky et al. (*Random Struct Algorithms*, 2015), we show how to deal with dependencies induced by the recent tree MIS algorithms when they are executed on bounded arboricity graphs. Specifically, we analyze a version of the tree MIS algorithm of Barenboim et al. and show that it runs in $O(\text{poly}(\alpha) \cdot \sqrt{\log n \cdot \log\log n})$ rounds in the $\mathcal{CONGEST}$ model for graphs with arboricity α.

While the main thrust of this paper is the new probabilistic analysis via read-k inequalities, we point out that for small values of α, this algorithm is faster than the MIS algorithm of Barenboim et al. specifically designed for bounded arboricity graphs. In this context, it should be noted that recently (in SODA 2016) Ghaffari presented a novel distributed MIS algorithm for general graphs that runs in $O(\log \Delta) + 2^{O(\sqrt{\log\log n})}$ rounds and a corollary of this algorithm is an $O(\log \alpha + \sqrt{\log n})$-round MIS algorithm on graphs with arboricity α.

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

Keywords and phrases Bounded Arboricity Graphs, CONGEST model, Luby's Algorithm, Maximal Independent Set, Read-k Inequality

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.9

* A version of this work appeared as a brief announcement in PODC 2016 [11].
† This work is supported in part by National Science Foundation grant CCF-1318166.

1 Introduction

A set of nodes in a graph is said to be *independent* if no two nodes in the set are adjacent. A *maximal independent set (MIS)* is an independent set that is maximal with respect to inclusion. Computing an MIS is a fundamental problem in distributed computing because it nicely captures the essential challenge of symmetry breaking and also for its myriad applications to other problems. The fastest algorithm for MIS is a simple, randomized algorithm discovered more than 25 years ago, independently by several researchers [1, 9, 7]. This algorithm computes an MIS for an n-node graph in $O(\log n)$ communication rounds with high probability (whp), i.e., with probability at least $1 - 1/n$. The essence of this algorithm is that in each round, each still-active node tentatively joins the MIS with some probability and then either backs off from this choice or makes it permanent depending on whether neighboring nodes have made conflicting choices. Following popular usage, we refer to this as *Luby's algorithm*. More recently, Métivier et al. [10] proposed a variant of Luby's algorithm in which in each round, each still-active node v picks a *priority*, a real number $r(v)$ uniformly at random from $[0, 1]$ and joins the MIS if $r(v)$ is greater than the priorities chosen by all neighbors. This algorithm also runs in $O(\log n)$ rounds whp [10][1].

In PODC 2011, Lenzen and Wattenhofer [8] showed that an MIS in an n-node *unoriented* tree can be computed in $O(\sqrt{\log n} \cdot \log\log n)$ rounds whp. Note that if a tree is consistently oriented (i.e., the tree is rooted at an arbitrary node and all nodes know their parent with respect to this root) then an MIS can be computed in $O(\log^* n)$ rounds using the deterministic coin tossing technique of Cole and Vishkin [4]. The first phase of the Lenzen-Wattenhofer algorithm is just the algorithm of Métivier et al. and in a sense all the important hard work happens in this phase. The running time analysis of the algorithm is sophisticated and depends critically on the fact that the tree structure ensures that there are very few dependencies among probabilistic events in the algorithm. There have been previous sublogarithmic-round MIS algorithms for special graph classes (e.g., the $O(\log^* n)$-round MIS algorithm on growth-bounded graphs [13]), but not for graphs that can have arbitrarily large independent sets in neighborhoods. Thus, in a sense, the Lenzen-Wattenhofer MIS result is a breakthrough because it shows that MIS can be computed in sublogarithmic rounds even in settings where neighborhoods can have arbitrarily many independent nodes. More recently in FOCS 2012, Barenboim et al. [2, 3] presented a tree MIS algorithm (similar to the Lenzen-Wattenhofer algorithm) that runs in $O(\sqrt{\log n \cdot \log\log n})$ rounds whp, improving the running time of the Lenzen-Wattenhofer algorithm slightly. This tree MIS algorithm also uses the algorithm of Métivier et al. to do a significant portion of the work.

A natural question that arises from the analyses of these tree MIS algorithms is whether the algorithms and analyses can be extended to *bounded arboricity graphs*. Lenzen and Wattenhofer raise this question at the end of the "Introduction" section in their paper [8]. A graph G is said to have *arboricity* α if α is the minimum number of forests that the edges of G can be partitioned into. From this it follows that the edges of a graph with arboricity α can be oriented in such a manner that each node has at most α outgoing edges. Clearly, forests have arboricity 1, but the family of graphs with constant arboricity is quite rich and includes all planar graphs, graphs with constant treewidth, graphs with constant genus, family of graphs that exclude a fixed minor, etc. Unfortunately, the Lenzen-Wattenhofer

[1] In fact, Algorithm A in Luby's 1986 paper [9] is essentially identical to the algorithm of Métivier et al., the only difference being that in Luby's Algorithm, vertices choose priorities from the range $\{1, 2, \ldots, n^4\}$. What we refer to as Luby's algorithm above appears as Algorithm B in Luby's paper.

analysis and the Barenboim et al. analysis runs into trouble for graphs with even *constant arboricity* because of the nature of dependencies between probabilistic events in the algorithm. The issue is common to both algorithms because it arises in the portions of the algorithms that rely on the algorithm of Métivier et al.

The source of the difficulty can be explained as follows. Even though these algorithms run on unoriented trees, *for the purposes of analysis* it can be assumed that the input tree is rooted at an arbitrary node. Because the graph is a tree, probabilistic events at children of a node v are essentially independent, the only slight dependency being caused by the interaction via their parent, namely v. For graphs with arboricity greater than 1 the dependency structure among the probabilistic events can be much more complicated. Suppose (for the purposes of the analysis) that we orient the edges of an arboricity-α graph such that each node has at most α out-neighbors. Let us call the out-neighbors of a node v its *parents* (denoted $\texttt{Parent}(v)$) and the in-neighbors, its *children* (denoted $\texttt{Child}(v)$). For a node v, consider the set $\texttt{Child}(v)$ and the dependencies among probabilistic events at nodes in $\texttt{Child}(v)$. The events we are referring to are of the type "w joins the MIS" or "a neighbor of w joins the MIS" for $w \in \texttt{Child}(v)$. Even though each node has at most α parents, a node $w \in \texttt{Child}(v)$ may share children with every other node in $\texttt{Child}(v)$ and as a result there could be dependencies between events at w and events at any of the other nodes in $\texttt{Child}(v)$. Thus it is not clear how to take advantage of the structure of bounded arboricity graphs in order to mimic the analysis in [8, 2, 3].

The main purpose of this paper is to show that recent results on *read-k families of random variables* deal with roughly this type of dependency structure and therefore provide a new approach to analyzing algorithms in the style of Métivier et al. with more complicated dependency structure. Using analysis based on *read-k inequalities* (see the next section), we show that the tree MIS algorithms of Lenzen and Wattenhofer [8] and Barenboim et al. [3, 2] work for bounded arboricity graphs as well. We believe that this analytical tool may be new to the distributed computing community, but will prove useful for the analysis of randomized distributed algorithms in general.

1.1 Read-k Inequalities

We now define a *read-k family* of random variables. Let $\{Y_j \mid 1 \leq j \leq n\}$ be a set of random variables such that each random variable Y_j is a function of some subset of the set of independent random variables $\{X_i \mid 1 \leq i \leq m\}$. For each $1 \leq j \leq n$, let $P_j \subseteq \{1, 2, \ldots, m\}$, let f_j be a boolean function of $\{X_i \mid i \in P_j\}$, and define $Y_j := f_j((X_i)_{i \in P_j})$. The collection of random variables Y_j is called a *read-k family* if every $1 \leq i \leq m$ appears in at most k of the P_j's. In other words, each X_i is allowed to influence at most k of the Y_j's. Note that the Y_j's can have a complicated dependency structure amongst themselves – it is their dependency on the X_i's that is bounded. For example, the dependency graph of the Y_j's can even be a clique!

We are now ready to state the first of the two read-k inequalities from Gavinsky et al. [5] that we use. This inequality provides a bound on the conjunction of a collection of events whose indicator variables form a read-k family.

▶ **Theorem 1** (Theorem 1.2, [5]). *Let $Y_1, Y_2, \cdots, Y_n$ be a family of read-k indicator variables with $\Pr[Y_i = 1] = p$. Then, $\Pr[Y_1 = Y_2 = \cdots = Y_n = 1] \leq p^{n/k}$.*

If the Y_j's were independent, then the probability that $Y_1 = Y_2 = \cdots = Y_n = 1$ would simply be p^n. Thus Theorem 1 is essentially saying that the read-k family structure of the

dependencies among the Y_j's allows us to obtain an upper bound on the probability that is an exponential factor $1/k$ worse than what is possible had the Y_j's been independent.

Gavinsky et al. [5] use Theorem 1 and information-theoretic arguments to derive the following tail inequality on the sum of indicator random variables that form a read-k family.

▶ **Theorem 2** (Theorem 1.1, [5]). *Let $Y_1, \cdots, Y_n$ be a family of read-k indicator variables with $\Pr[Y_i = 1] = p_i$. Define $p := \frac{1}{n}\sum_{i=1}^{n} p_i$ and $Y := \sum_{i=1}^{n} Y_i$. Then for any $\epsilon, \delta > 0$,*

$$\Pr(Y \leq (p-\epsilon)n) \leq \exp\left(-2\epsilon^2 \frac{n}{k}\right), \tag{1}$$

$$\Pr(Y \leq (1-\delta)E[Y]) \leq \exp\left(-\frac{\delta^2 E[Y]}{2k}\right). \tag{2}$$

Gavinsky et al. only state Form (1) of the tail inequality in their paper. But, Form (2) is more convenient for us and it is fairly routine to derive this from Form (1). See [14] for the derivation. Note that upper tail inequalities corresponding to (1) and (2) also exist, but have not been shown here for brevity. As in Theorem 1, these tail inequalities are also an exponential $1/k$ factor worse than corresponding Chernoff bounds that we might have used, had the Y_j's been independent. Gavinsky et al. [5] also point out that these tail inequalities are more general than those that can be obtained by observing that Y is a k-Lipschitz function and using standard Martingale-based arguments such as Azuma's inequality.

To see that the above tools are well-suited for analyzing algorithms in the style of Métivier et al. on bounded arboricity graphs, let us reconsider the situation described earlier. Consider a graph G with arboricity α and fix an arbitrary node v in G, and consider the set $\texttt{Child}(v)$ of children of v. For the moment, ignore edges among nodes in $\texttt{Child}(v)$ and also ignore the influence of parents (v and other parents) on nodes in $\texttt{Child}(v)$, thus focusing only on the children of nodes in $\texttt{Child}(v)$. For a node $w \in \texttt{Child}(v)$, let Y_w be an indicator variable for a probabilistic event at node w. Now suppose that Y_w depends on independent random choices made by w and its children. For example, Y_w could be a boolean variable indicating the event that the priority of w is larger than the priorities of children of w. This models the situation in the algorithm of Métivier et al. [10], where w joining the MIS depends on random real values (independently) chosen by w and its children. (Recall that we are ignoring parents for the moment.) The structure of an arboricity-α graph and the associated edge-orientation ensures that each node has at most α parents and therefore the random choice at each node can influence at most α of the Y_w's. Thus the set $\{Y_w \mid w \in \texttt{Child}(v)\}$ forms a read-α family and we can apply Theorems 1 and 2 to bound $Pr(\cap_w Y_w = 1)$ and to show that $\sum_w Y_w$ is concentrated around its expectation.

The above example illustrates the simplest application of read-k inequalities in our analysis. Somewhat surprisingly, we use read-k inequalities to evaluate probabilistic interactions between a node and its parents also. This may seem impossible to do given that a parent can have arbitrarily many children and thus a random choice at a parent can influence events at arbitrarily many children. However, in our algorithm nodes with extremely high degree opt out of the competition (temporarily) and this turns out to be sufficient to bound the number of children a parent can influence, leading to our use of read-k inequalities, with appropriate k, to analyze the interaction between nodes and their parents. Finally, our analysis also relies on interactions between a node and its grandchildren, leading to our use of read-$\Theta(\alpha^2)$ families as well.

1.2 Our Result

We apply a read-k-inequality-based analysis to the execution of the tree MIS algorithm of Barenboim et al. [3, 2] on bounded arboricity graphs. We could have chosen to analyze the tree MIS algorithm of Lenzen and Wattenhofer, but for reasons of exposition we use the algorithm of Barenboim et al. We present an algorithm that we call BoundedArbIndependentSet, which is essentially identical to the TreeIndependentSet algorithm of Barenboim et al. (Section 8, [3]), except for parameter values (which now depend on the arboricity α). Specifically, we show the following result.

▶ **Theorem 3.** *The tree MIS algorithm of Barenboim et al.* [3, 2] *(with appropriate parameter values) can be used to compute an MIS in the $\mathcal{CONGEST}$ model on the family of graphs with arboricity α in $O(poly(\alpha) \cdot \sqrt{\log n \cdot \log\log n})$ rounds, whp.*

This result can also be seen as an improvement over the MIS result on bounded arboricity graphs due to Barenboim et al. [3, 2]. In their paper, Barenboim et al. have a separate algorithm (distinct from their tree MIS) algorithm that computes an MIS on graphs with arboricity α in $O(\log^2 \alpha + \log^{2/3} n)$ rounds. The dependency on n of the running time of our algorithm is asymptotically better, implying that for small α (i.e., $\alpha = O(\log^c n)$ for a small enough constant c) our algorithm is asymptotically faster. In our subsequent calculations the degree of polynomial in α in the running time comes out to be 9. It is not difficult to reduce this degree, but it does seem difficult with the current algorithm to improve the dependency on α to something better than a polynomial and to replace the multiplication between the poly(α)-term and the $\sqrt{\log n \cdot \log\log n}$-term by an addition.

Recently, in SODA 2016 Ghaffari has presented a novel MIS algorithm [6] that runs in $O(\log \Delta) + 2^{O(\sqrt{\log\log n})}$ rounds on any n-vertex graph with maximum degree Δ. In Luby's MIS algorithm, a node's "desire" to join the MIS is a simple function of its degree with respect to the still-active nodes in the graph. In Ghaffari's MIS algorithm each node explicitly maintains a *desire-level* that is initially set to 1/2, but is updated in each iteration depending on the aggregate desire-level of nodes its neighborhood. Using techniques from [3, 2], Ghaffari obtains, as a corollary of this main result, an $O(\log \alpha + \sqrt{\log n})$-round MIS algorithm for n-vertex graphs with arboricity α. This of course dominates the round complexity our algorithm for all values of α and n. Thus the main contribution of this paper is not the fastest distributed MIS algorithm for bounded arboricity graphs, but it is (i) introducing the use of read-k inequalities for the analysis of randomized distributed algorithms and (ii) showing that recent tree MIS algorithms are effective for bounded arboricity graphs as well, but need more sophisticated analysis.

2 MIS Algorithm for Bounded Arboricity Graphs

We start by presenting an algorithm that we call BoundedArbIndependentSet, which is essentially identical to the TreeIndependentSet algorithm of Barenboim et al. (Section 8, [3]), except for parameter values (Θ, Λ, ρ_k) which now depend on α as well. We emphasize this point because we are essentially analyzing the TreeIndependentSet algorithm (via a new approach based on the read-k inequalities), but with bounded arboricity graphs as input.

The algorithm (see **Algorithm 1**) begins by initializing two sets I and B as empty. I denotes the set of nodes which have joined the MIS and B will store a set of so-called "bad" nodes. As nodes join I and B, they exit the algorithm, i.e., become *inactive*. In addition, neighbors of nodes in I also exit the algorithm and become inactive. We use V_{IB} to

Algorithm 1: BoundedArbIndependentSet(Graph G):

1: Initialize sets $I, B \subseteq V(G)$: $I \leftarrow \phi$; $B \leftarrow \phi$
2: **for** *each scale* k *from 1 to* $\Theta := \left\lfloor \log\left(\frac{\Delta}{1176 \cdot 16\alpha^{10} \ln^2 \Delta}\right)\right\rfloor$ **do**
Initialize $\rho_k \leftarrow 8 \ln \Delta \cdot \Delta/2^{k+1}$
2(a) Execute $\Lambda := \lceil p \cdot 8\alpha^2(32\alpha^6 + 1) \cdot \ln(260\alpha^4 \ln^2 \Delta)\rceil$ times
- Each node $v \in V_{IB}$ chooses a priority $r(v)$:

$$r(v) \leftarrow \begin{cases} 0, \text{ if } \deg_{IB}(v) > \rho_k \\ \text{a real in} (0,1) \text{ chosen uniformly at random otherwise} \end{cases}$$

- $I \leftarrow I \cup \{v \in V_{IB} | r(v) > max\{r(w) | w \in \Gamma_{IB}(v)\}\}$
- $V_{IB} \leftarrow V_{IB} \setminus (I \cup \Gamma_{IB}(I))$

2(b) Each node v is marked "bad" if $|\{w \in \Gamma_{IB}(v) | deg_{IB}(w) > \Delta/2^k + \alpha\}| > \Delta/2^{k+2}$
$B \leftarrow B \cup \{v \in V_{IB} \mid v \text{ is marked "bad"}\}$
$V_{IB} \leftarrow V_{IB} \setminus B$
end
3: **return** (I, B)

denote the set of nodes which are currently active. Let $\Gamma_{IB}(u)$ represent the neighborhood of a node u restricted to nodes in V_{IB}. Let $deg_{IB}(u)$ denote $|\Gamma_{IB}(u)|$. Similarly, for any subset $S \subseteq V$ of nodes, let $\Gamma_{IB}(S)$ denote $\cup_{u \in S} \Gamma_{IB}(u)$. The algorithm proceeds in $\Theta := \left\lfloor \log\left(\frac{\Delta}{1176 \cdot 16\alpha^{10} \ln^2 \Delta}\right)\right\rfloor$ *scales*. For any scale k, $1 \leq k \leq \Theta$, a node in V_{IB} that has degree more than $\Delta/2^k + \alpha$ is called a *high degree* node for that scale. In each scale, we start by performing $O(\alpha^8(\log \alpha + \log\log \Delta))$ *iterations* of the Métivier et al. MIS algorithm [10]. The exact number of iterations is $\lceil p \cdot 8\alpha^2(32\alpha^6 + 1) \cdot \ln(260\alpha^4 \ln^2 \Delta)\rceil$ and denoted by the parameter Λ, where p is a large enough constant whose value will be fixed later.

In a single iteration, every node $v \in V_{IB}$ chooses a real number $r(v) \in [0, 1)$ called a *priority*. If v has more than $\rho_k := 8 \ln \Delta \cdot \Delta/2^{k+1}$ neighbors in any iteration, its priority is (deterministically) set to 0, otherwise, it chooses a priority uniformly at random in $(0, 1)$. In any iteration, a node u is called *competitive*, if $r(u)$ is chosen randomly in that iteration. If in an iteration, v chooses a priority greater than the priority of any node in its neighborhood in V_{IB}, it joins I. After each iteration, nodes in I and neighbors of these nodes (i.e., $\Gamma_{IB}(I)$) are removed from V_{IB}. If, after Λ iterations in the current scale, a node $v \in V_{IB}$ has more than $\Delta/2^{k+2}$ high-degree neighbors then it is designated a "bad" node and added to the set B It is worth emphasizing the fact that this algorithm has no access to an edge-orientation or a forest-decomposition of the given α-arboricity graph. We use the existence of an edge-orientation extensively in our analysis, but it plays no role in the algorithm.

2.1 Finishing Up

The algorithm returns an independent set I (which need not be maximal), and a set B of "bad" nodes. Also, the set V_{IB} need not be empty at the end of the algorithm and so after Algorithm BoundedArbIndependentSet has completed, we still need to process the sets B and V_{IB}.

Our main contribution in this paper is a new analysis of BoundedArbIndependentSet that culminates in Theorem 10, showing that any node joins B with probability at most $1/\Delta^{2p}$. (Here p is the constant that is used in determining Λ, the number of iterations of BoundedArbIndependentSet.) The fact that each node joins B with very low probability

implies (as shown by Barenboim et al. [3] and restated in Lemma 11) that with high probability all connected components in the graph induced by B are small. These components induced by B can be processed in parallel, with each component being processed by a deterministic algorithm (since each component is small).

Nodes that remain in V_{IB} have the property that they do not have too many high degree neighbors. Otherwise, they would have been placed in B. Thus V_{IB} can be partitioned into two sets V_{hi} and V_{low} such that the graphs induced by each of these sets has small maximum degree. Then, by using an alternate MIS algorithm that finishes quickly as a function of the maximum degree, we process nodes in V_{hi} and V_{low} (one set after the other) to complete the MIS computation. All these steps that "finish up" the algorithm run in the $\mathcal{CONGEST}$ model and we describe these in greater detail in Section 3.3.

It is immediate that $O(\alpha^8(\log\alpha + \log\log\Delta)\cdot\log\Delta)$ is the number of rounds it takes to complete algorithm BoundedArbIndependentSet. The rest of the algorithm (described informally above and in more detail in Section 3.3) takes an additional $O(\alpha^2 + \log\Delta + \log\log n\cdot\log\alpha + \alpha\cdot\log^* n)$ rounds whp. To get a round complexity bound that is exclusively in terms of n and α, we use a degree-reduction result of Barenboim et al. (Theorem 7.2 [3]) that runs in $O(\sqrt{\log n\cdot\log\log n})$ rounds in the $\mathcal{CONGEST}$ model and yields a graph with maximum degree at most $\alpha\cdot 2^{\sqrt{\log n/\log\log n}}$. We use this degree reduction as a preprocessing step and use $\Delta = \alpha\cdot 2^{\sqrt{\log n/\log\log n}}$ in the rest of the algorithm.

▶ **Theorem 4.** *Using* BoundedArbIndependentSet *we can compute an MIS on a graph with arboricity α in $O\left(\alpha^8(\log\alpha + \log\log\Delta)\cdot\log\Delta + \log\log n\log\alpha\right)$ rounds whp. This leads to an algorithm that computes an MIS on a graph with arboricity α in $O(\alpha^9\sqrt{\log n\cdot\log\log n})$ rounds whp.*

3 Analysis of BoundedArbIndSet

We start with an overview of our analysis. At a high level, the organization of our analysis is similar to the analysis of TreeIndependentSet. The analysis is centered around showing that the following invariant is maintained (at the end of each scale) at every active node, with sufficiently high probability.

> Invariant: At the end of scale k, for all $v \in V_{IB}$,
>
> $$\left|\{w \in \Gamma_{IB}(v) | \deg_{IB}(w) > \Delta/2^k + \alpha\}\right| \leq \Delta/2^{k+2}$$

The Invariant bounds the number of high degree neighbors a node has after k scales of the algorithm. In a sense the Invariant is trivially satisfied by design; nodes that do not satisfy the Invariant after Λ iterations in Scale k are simply placed in the set B (of "bad" nodes) in Step 2(b) of the algorithm. Of course, we have to later on deal with the nodes in B somehow and so we cannot simply place all nodes in B and claim to have satisfied the Invariant! Let N denote the set on the left-hand side of the Invariant above. The goal then is to show that, with probability at least $1 - 1/\Delta^2$, in Scale k, either v becomes inactive (Lemma 8) or the size of N falls to $\Delta/2^{k+2}$ or less (Lemma 9). Showing this leads to Theorem 10 which claims that that after Scale k, each active node satisfies the invariant with probability at least $1 - 1/\Delta^2$ and is therefore placed in B with probability at most $1/\Delta^2$.

Unlike in the analysis of Algorithm TreeIndependentSet, for bounded arboricity graphs, the proof of Theorem 10 has to deal with seemingly complicated dependencies among

probabilistic events that the algorithm depends on. Our main contribution in this paper is to show that all of these dependencies can be quite naturally analyzed via read-k inequalities (with different values of the parameter k). So first, in Section 3.1, we use read-k inequalities to analyze three key probabilistic events pertaining to the progress of the algorithm. Later on we show how the occurrence of these probabilistic events with sufficiently high probability holds the key to proving Theorem 10.

Notation: For the purposes of the analysis we fix an edge orientation of the given arboricity-α graph such that each node has at most α out-neighbors (parents). We use $\texttt{Parent}_{IB}(v)$ to denote the set of currently active parents of node v and $\texttt{Child}_{IB}(v)$ to denote the set of currently active children of node v. For any subset S of nodes, we use $\Delta_{IB}(S)$ to denote $\max_{v \in S} deg_{IB}(v)$.

3.1 Read-k Inequalities in Action

In this section, we analyze via read-k inequalities, three key probabilistic events whose occurrence (with sufficient probability) ensures rapid progress of our algorithm. The first event concerns the interaction between nodes and their children and the second concerns the interaction between nodes and their parents. The third event is more complicated and it concerns the interaction between nodes and their children, their children's children (i.e., *grandchildren*) and their children's other parents (i.e., *co-parents*). To be more specific, let us fix a Scale k and an iteration within that scale. Let $M \subseteq V_{IB}$ be an active subset of nodes just before the start of the iteration under consideration. The three probabilistic events we analyze can be informally described as follows. For Events (1) and (2), we assume that all nodes in M have degree at most ρ_k and are therefore competitive.

Event (1) Among the set of nodes M, there exists a node whose priority is larger than the priority of all its children.
Event (2) Suppose that M is sufficiently large. Then a large fraction of the nodes in M have priority greater than priorities of all their parents.
Event (3) Suppose that every node in M has sufficiently high degree. Then a large fraction of the nodes in M become inactive due to their children joining the MIS.

The simplest approach to analyzing these events is to decompose each event into sub-events centered at each of the nodes in M and then apply a tail inequality such as the Chernoff bound. The difficulty with this approach of course is the lack of independence among the sub-events at nodes in M. However, as we discuss below and then show later, each of these collections of sub-events can be analyzed using a read-k inequality with different values of the parameter k.

Event (1) (see Theorem 5 and Figure 1(A)) can be viewed as the complement of the event in which every node in M has a child with greater priority. This latter event is a conjunction of events, E_x for $x \in M$, where $E_x \equiv r(x) < \max\{r(y) \mid y \in \texttt{Child}_{IB}(x)\}$. However, for nodes $x, x' \in M$, E_x and $E_{x'}$ need not be independent because x and x' may share children. Nevertheless, since a child can have at most α parents, the collection $\{E_x \mid x \in M\}$ of events has a dependency structure that forms a read-α family and we can analyze Event (1) by applying the read-α conjunction inequality (Theorem 1).

We can attempt to analyze Event (2) (see Theorem 6 and Figure 1(B)) in a similar manner. For each $x \in M$, let $F_x \equiv r(x) > \max\{r(y) \mid y \in \texttt{Parent}_{IB}(x)\}$. However, dependencies

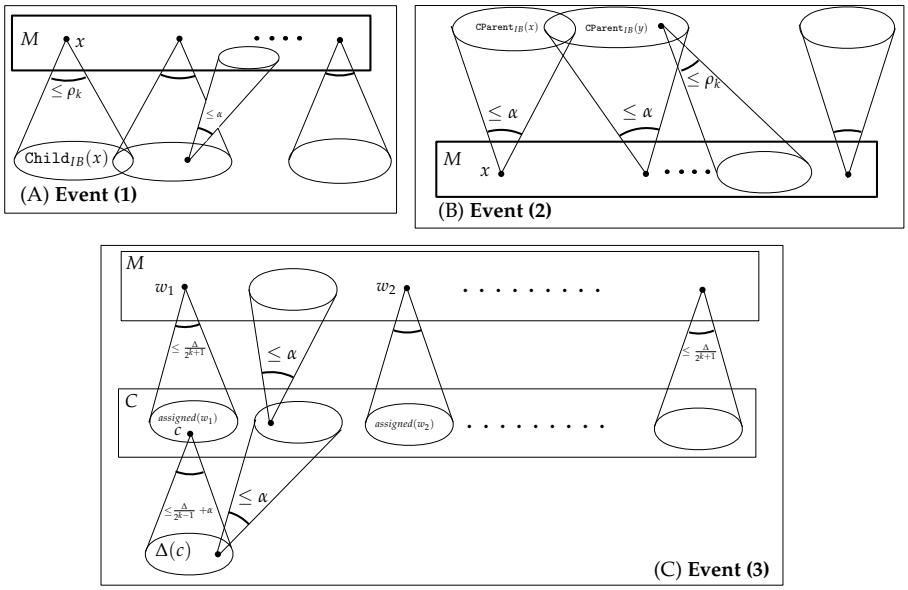


Figure 1 (A) shows the application of read-α inequalities to lower bound the probability of some node $x \in M$ having priority greater than priorities of all its children. (B) shows the application of a read-ρ_k inequality to prove that with sufficient probability a large fraction of the nodes in M have priorities greater than priorities of parents. (C) shows the application of a read-$\Theta(\alpha^2)$ inequality to prove that with sufficient probability, a large fraction of the nodes in M are eliminated by children joining the MIS.

among the events $\{F_x \mid x \in M\}$ are harder to deal with because a node can be the parent of arbitrarily many nodes in M and thus possibly affect all nodes in M. However, recall that a node with degree greater than ρ_k does not participate in the competition to join the MIS (it simply sets its priority to 0). Thus, if M is significantly larger than ρ_k then a competitive node can only be the parent of a small fraction of nodes in M. Thus the events $\{F_x \mid x \in M\}$ have a read-ρ_k dependency structure and we can apply a read-ρ_k tail inequality to analyze this event.

Event (3) (see Theorem 7 and Figure 1(C)) pertains to the elimination of nodes in M due to children of these nodes joining the MIS. Following the approach used to analyze Events (1) and (2), we consider events G_x for $x \in M$ where G_x is the event that some child of x joins the MIS. Whether a child w of $x \in M$ joins the MIS, depends on the priorities at w and neighbors of w. Specifically, G_x depends on the priority of x and the priorities of children of x, grandchildren of x, and co-parents of x. As a result, the dependencies among the events $\{G_x \mid x \in M\}$ are much more complicated to analyze and cannot be directly analyzed using read-k inequalities. To get around this problem, we apply the analysis of Event (2) (Theorem 6) to show that with sufficiently high probability, a substantial fraction of the children of $x \in M$ have priorities greater than all their parents. We then condition on this event and only focus on such children (denoted $\texttt{Child}'_{IB}(x)$) of each $x \in M$. Now let us redefine G_x as the event that some node in $\texttt{Child}'_{IB}(x)$ has priorities greater than all its *children*. Note that if a node $w \in \texttt{Child}'_{IB}(x)$ has priority greater than its children, it will join the MIS (thereby eliminating x) since its priority is known to be greater than the priorities of parents. Thus, if the redefined G_x occurs, then x is eliminated. Now note that each G_x depends on the priority of x, priorities of children of x and priorities of grandchildren of x. Given that each node has at most α parents and α^2 grandparents, we can see that

the collection $\{G_x \mid x \in M\}$ forms a read-$\alpha(\alpha+1)$ family, allowing us to use read-$\alpha(\alpha+1)$ inequalities to analyze Event (3). In the three theorems that follow, we formally describe and analyze Events (1)-(3).

▶ **Theorem 5** (Event (1)). *For some Scale k and some iteration in this scale, let $M \subseteq V_{IB}$ be an independent subset of nodes that are active just before the start of the iteration. Further suppose that $\Delta_{IB}(M) \leq \rho_k$. Then, with probability at least $1 - \left(1 - \frac{1}{\Delta_{IB}(M)}\right)^{|M|/(2\alpha^2)}$, some node in M will choose a priority greater than the priorities of all of its children. This holds even when we condition on all nodes in M having priorities greater than their parents' priorities.*

Proof. Since the graph induced by V_{IB} has arboricity at most α, there exists an independent set[2] $M_{ind} \subseteq M$ such that $|M_{ind}| \geq |M|/2\alpha$. Let $x^* \in M_{ind}$ be a node that chooses a priority $r(x^*)$ greater than all its children, i.e., $r(x^*) > \max\{r(y) \mid y \in \texttt{Child}_{IB}(x^*)\}$ in the iteration being considered. We now calculate the probability that such an x^* exists. For each node $x \in M_{ind}$, let E_x denote the event $r(x) < \max\{r(y) \mid y \in \texttt{Child}_{IB}(x)\}$ and let Y_x be the indicator variable for E_x. We now argue that the collection of random variables $\{Y_x \mid x \in M_{ind}\}$ forms a read-α family. See Figure 1(A).

Read-α family. Each Y_x is a function of independent random variables, namely the priority $r(x)$ and the priorities of children of x, i.e., $\{r(y)|y \in \texttt{Child}_{IB}(x)\}$. Thus a priority $r(w)$ can only influence random variables Y_x, where x is a parent of w and this means that each priority can influence at most α elements in $\{Y_x \mid x \in M_{ind}\}$. Therefore the set of random variables $\{Y_x \mid x \in M_{ind}\}$ forms a read-α family.

Now note that $Y_x = 0$ corresponds to $r(x)$ being larger than $r(y)$ for all $y \in \texttt{Child}_{IB}(x)$. Therefore, $Pr(Y_x = 0) \geq \frac{1}{\Delta_{IB}(M)}$, implying that $Pr(Y_x = 1) \leq \left(1 - \frac{1}{\Delta_{IB}(M)}\right)$. Note that this depends on the fact that $deg_{IB}(x) \leq \rho_k$ and x is competitive. Using this bound and the conjunctive read-α inequality in Theorem 1, we see that $Pr(\cap_{x \in M_{ind}} Y_x = 1) \leq (1 - 1/\Delta_{IB}(M))^{(|M|/2\alpha)\cdot(1/\alpha)}$. Thus the probability that there exists an $x^* \in M_{ind}$ for which E_{x^*} holds is as claimed. ◀

▶ **Theorem 6** (Event (2)). *For some scale k and some iteration in this scale, let $M \subseteq V_{IB}$ be a subset of nodes that are active just before the start of the iteration. Further suppose that $\Delta(M) \leq \rho_k$ and $|M| > 64\alpha^2 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$. Then, at the end of the iteration, with probability at least $(1 - 1/\Delta^4)$, the number of nodes in M that choose a priority greater than their parents is more than $|M|/2\alpha$.*

Proof. The probability that a node in M chooses a priority greater than its parents is equal to the probability that it chooses a priority greater than its competitive parents. (Recall that a non-competitive node has degree more than ρ_k and it deterministically sets its priority to 0.) Let $\texttt{CParent}_{IB}(u)$ denote the set of current competitive parents of a node u.
For any node $u \in M$, let F_u denote the event $r(u) > \max(\{r(y)|y \in \texttt{CParent}_{IB}(u)\})$ and let X_u be the indicator variable for F_u. Let $X = \sum_{u \in M} X_u$ be the random variable representing the number of nodes in M whose priorities are greater than priorities of their parents. Since each node can have at most α parents and since $deg_{IB}(x) \leq \rho_k$, $Pr(X_u = 1) = E[X_u] \geq 1/\alpha$ and $E[X] \geq |M|/\alpha$. We would now like to show that X is concentrated about its expectation,

[2] By repeatedly adding a vertex with degree at most α to the independent set, we can see that there is an independent set of size at least $|M|/(\alpha+1)$ in the graph induced by V_{IB}.

but cannot use Chernoff bounds because the variables $\{X_u \mid u \in M\}$ are not mutually independent. Again, a read-k inequality comes to the rescue and we first show that the set of variables $\{X_u \mid u \in M\}$ forms a read-ρ_k family.

Read-ρ_k family. Each X_u is a function of independent random variables, namely its own priority $r(u)$ and the priorities of its competitive parents. Since any competitive node $w \in V_{IB}$ has degree at most ρ_k, a priority $r(w)$ influences at most ρ_k X_u's. Therefore, $\{X_u \mid u \in M\}$ forms a read-ρ_k family and we can apply the read-ρ_k tail inequality in Theorem 2 (Form (1)) to establish the concentration of X about its expectation as follows:

$$Pr(X \leq (1/\alpha - 1/2\alpha) \cdot |M|) \leq \exp\left(-2(1/4\alpha^2) \cdot \frac{|M|}{\rho_k}\right).$$

Since $|M| > 64\alpha^2 \ln^2 \Delta \cdot \Delta/2^{k+1}$,

$$Pr(X \leq |M|/2\alpha) \leq \exp\left(-2(1/4\alpha^2) \cdot \frac{\Delta\alpha^2(64\ln^2\Delta)/2^{k+1})}{\Delta(8\ln\Delta)/2^{k+1}}\right) \leq \exp(-4\ln\Delta).$$

Thus, the probability that $X > |M|/2\alpha$ is at least $(1 - 1/\Delta^4)$. ◀

▶ **Theorem 7** (Event (3)). *For some scale k and some iteration in this scale, let $M \subseteq V_{IB}$ be a subset of nodes that are active just before the start of the iteration. Further suppose that $|M| > \Delta/2^{k+2}$ and $\deg_{IB}(w) > \Delta/2^k + \alpha$ for all nodes $w \in M$. Then with probability at least $(1 - 1/\Delta^3)$ at least $|M|/8\alpha^2(32\alpha^6 + 1)$ nodes in M are eliminated in the iteration.*

Proof. Applying the Invariant, at the end of the scale $k-1$, we see that each node w in M has at most $\Delta/2^{k+1}$ neighbors with degree more than $\Delta/2^{k-1} + \alpha$. Therefore, w has at least $\deg_{IB}(w) - \Delta/2^{k+1} - \alpha > \Delta/2^{k+1}$ children with degree at most $\Delta/2^{k-1} + \alpha$. For the purposes of this theorem, we will refer to these nodes as *low-degree* children.

We now construct a set C, that consists of low-degree children of nodes in M. Consider nodes in M in some arbitrary order $w_1, w_2, \ldots, w_{|M|}$. For w_1, pick $\Delta/2^{k+1}$ low-degree children from among the more than $\Delta/2^{k+1}$ such children that it has. These nodes are said to be *covered* and *assigned* to w_1. For each node w_i, $1 < i \leq |M|$, let c_i be the number of low-degree children of w_i that have already been covered. If c_i is at least $\Delta/2^{k+1}$, we do nothing. Otherwise, pick $(\Delta/2^{k+1} - c_i)$ low-degree children of w_i arbitrarily and declare these nodes covered and assign them to w_i. Let C be the set of all covered nodes at the end of this procedure.

Now note that each node in $w_i \in M$ has at least $\Delta/2^{k+1}$ children in C and at most $\Delta/2^{k+1}$ of these children are assigned to it. See Figure 1(C). Since each node in C has at most α parents in M, C has size at least $\frac{|M|}{\alpha} \cdot \frac{\Delta}{2^{k+1}}$. Note that since $|M| > \Delta/2^{k+2}$ and the maximum value of the scale index k is bounded above by $\log\left(\frac{\Delta}{1176 \cdot 16\alpha^{10}\ln^2\Delta}\right)$, using a little algebra we see that $|M|$ is more than $64\alpha^4\ln^2\Delta$ and therefore $|C|$ is more than $64\alpha^3\ln^2\Delta \cdot \Delta/2^{k+1}$ for all values of k. Then, applying Theorem 6 on the set C (since it is large enough), we see that with probability at least $1 - 1/\Delta^4$, more than $|C|/2\alpha$ nodes in C choose a priority higher than their parents' priority. Let C' denote the subset of nodes in C that have chosen a priority higher than priorities of their parents. Let E denote the event that $|C'| > |C|/2\alpha$. (Thus, E happens with probability at least $1 - 1/\Delta^4$.) We now condition on event E and using a simple averaging argument we show that there are a significant fraction of the nodes in M, each having sufficiently many children in C'. This is stated in the claim below. The point of this is that for such nodes in M to be eliminated, it would suffice for a child in C' to have priority larger than priority of its children – since nodes in C' already have priority more than priorities of parents.

▶ **Claim.** *Conditioned on E, there are at least $|M|/4\alpha^2$ nodes in M that have more than $\frac{1}{2\alpha^3} \cdot \frac{\Delta}{2^{k+1}}$ children each in C'.*

Proof. Let O be the subset of nodes in M that have at most $\frac{1}{2\alpha^3} \cdot \frac{\Delta}{2^{k+1}}$ children in C'. To calculate a lower bound on $|M| - |O|$, we will try to cover nodes in C' using O and $M \setminus O$. Each node in O is assigned at most $\frac{1}{2\alpha^3} \cdot \frac{\Delta}{2^{k+1}}$ nodes in C' and each node in $M \setminus O$ is assigned at most $\Delta/2^{k+1}$ nodes in C'. Thus,

$$(|M| - |O|) \cdot \frac{\Delta}{2^{k+1}} + |O| \left(\frac{1}{2\alpha^3} \cdot \frac{\Delta}{2^{k+1}} \right) \geq |C'| \geq \frac{|C|}{2\alpha} \geq \frac{|M|}{2\alpha^2} \cdot \frac{\Delta}{2^{k+1}}.$$

Note that the second-last inequality above depends on the conditioning on event E. Manipulating this expression we get the following upper bound on $|O|$:

$$|O| \leq |M| \left(\frac{1 - 1/2\alpha^2}{1 - 1/2\alpha^3} \right).$$

Therefore,

$$|M| - |O| \geq |M| \left(1 - \frac{1 - 1/2\alpha^2}{1 - 1/2\alpha^3} \right) \geq \frac{|M|}{4\alpha^2}.$$

The last inequality above holds for all $\alpha \geq 2$. ◀

Let M' denote the subset of M of nodes each having at least $\frac{1}{2\alpha^3} \cdot \frac{\Delta}{2^{k+1}}$ children in C'. Thus the above claim shows that conditioned on event E, $|M'| \geq |M|/4\alpha^2$. Consider an arbitrary node $w \in M'$. Now note that $|\texttt{Child}_{IB}(w) \cap C'| \geq \frac{1}{2\alpha^3} \cdot \frac{\Delta}{2^{k+1}}$ and $\Delta(\texttt{Child}_{IB}(w) \cap C') \leq \Delta/2^{k-1} + \alpha$. This means that we can apply Theorem 5 to the set $\texttt{Child}_{IB}(w) \cap C'$ and conclude that the probability that some node in $\texttt{Child}_{IB}(w) \cap C'$ will have priority greater than the priorities of all its children is at least

$$1 - \left(1 - \frac{1}{\Delta/2^{k-1} + \alpha}\right)^{\Delta/(2\alpha^3 \cdot 2^{k+1}) \cdot (1/2\alpha^2)} \geq 1 - \exp\left(-\frac{2^{k-1}}{2\Delta\alpha} \cdot \frac{\Delta}{2^{k+1} 4\alpha^5} \right) > \left(1 - e^{-1/32\alpha^6}\right).$$

This last expression can be bounded below by $1/(32\alpha^6 + 1)$.

For any $w \in M'$, let G_w denote the event that some node $\texttt{Child}_{IB}(w) \cap C'$ has priority greater than the priorities of all its children. Let Z_w be the indicator variable for event G_w. By the above calculation we see that $Pr(Z_w = 1) \geq \frac{1}{32\alpha^6+1}$. Let $Z = \sum_{w \in M'} Z_w$. Note that if a node x in $\texttt{Child}_{IB}(w) \cap C'$ has priority greater than the priorities of children, then it joins the MIS since we already know that it has priority greater than the priorities of parents. Thus Z is a lower bound on the number of nodes in M' that are eliminated in this iteration of the algorithm. By linearity of expectation, we see that $E[Z] \geq \frac{|M'|}{32\alpha^6+1}$. We would now like to finish the proof of the theorem by showing that with sufficiently high probability, Z is at least one-half of its expectation. Unfortunately, the Z_w's are not mutually independent and we cannot use Chernoff tail bounds to show the concentration of Z about its expectation. Nevertheless we are able to show that the random variables $\{Z_w \mid w \in M'\}$ form a read-$\alpha(\alpha + 1)$ family and exploit this structure to show the tail bound we need.

Read-$\alpha(\alpha + 1)$ family. Note that each Z_w is a function of $r(w)$, priorities of children of w, and priorities of grandchildren of w. It is important to note here that parents of w and co-parents of w have no role to play in determining the value of Z_w. Since the graph has arboricity α, for any node x, $r(x)$ may influence at most $\alpha(\alpha + 1)$ of the variables in

$\{Z_w \mid w \in M'\}$. Using the read-$\alpha(\alpha+1)$ tail inequality in Theorem 2 (Form (2)), we see that:

$$Pr(Z < E[Z]/2) \leq \exp\left(-\frac{E[Z]}{8\alpha(\alpha+1)}\right) \leq \exp\left(-\frac{|M'|}{8\alpha(\alpha+1)(32\alpha^6+1)}\right).$$

Now we condition on the event E and use the fact that conditioned on E, $|M'| \geq |M|/4\alpha^2$ and $E[Z] \geq |M|/(4\alpha^2(32\alpha^6+1))$ to obtain:

$$Pr\left(Z < \frac{|M|}{8\alpha^2(32\alpha^6+1)}\middle| E\right) \leq \exp\left(-\frac{|M|}{32\alpha^3(\alpha+1)(32\alpha^6+1)}\right).$$

According to the hypothesis of the theorem, $|M| > \Delta/2^{k+2}$ and we know that the maximum value of the scale index k is bounded above by $\log\left(\frac{\Delta}{1176\cdot 16\alpha^{10}\ln^2\Delta}\right)$. Using a little algebra we see that $|M|$ is more than $1176 \cdot 4\alpha^{10}\ln^2\Delta$ for all values of k. Therefore,

$$Pr\left(Z < \frac{|M|}{8\alpha^2(32\alpha^6+1)}\middle| E\right) \leq \exp\left(-\frac{1176\cdot 4\alpha^{10}\ln^2\Delta}{32\alpha^3(\alpha+1)(32\alpha^6+1)}\right) \leq \exp(-\ln^2\Delta).$$

Finally, noting that $Pr(E) \geq (1-1/\Delta^4)$, we see that

$$Pr\left(Z \geq \frac{|M|}{8\alpha^2(32\alpha^6+1)}\right) \geq (1-\exp(-\ln^2\Delta))\cdot(1-1/\Delta^4) \geq (1-1/\Delta^3).$$

Therefore, with probability at least $(1-1/\Delta^3)$, at least $|M|/8\alpha^2(32\alpha^6+1)$ fraction of the nodes in M are eliminated in each iteration. ◀

3.2 Proving the Invariant

In this section we show inductively that the Invariant holds after every scale. Suppose that the Invariant holds after Scale $k-1$ for any $k \geq 1$. (Note that "end of Scale 0" refers to the beginning of the algorithm.) Fix a node v and let $N = \{w \in \Gamma_{IB}(v) | \deg_{IB}(w) > \Delta/2^k + \alpha\}$ be the set of active *high-degree* neighbors of v at the beginning of Scale k. To establish that the Invariant holds after Scale k we will show that with sufficiently high probability either (i) v is eliminated in Scale k or (ii) $|N|$ shrinks to at most $\Delta/2^{k+2}$ by the end of Scale k. We consider two cases depending on the size of N and show that (i) holds when $|N|$ is large (Lemma 8) and (ii) holds when $|N|$ is smaller, but still bigger than $\Delta/2^{k+2}$ (Lemma 9). We note that this organizational structure of our overall proof is similar to the approach used by Barenboim et al. [3, 2]. Our main innovation and contribution appears in the previous section where we analyze, via read-k inequalities, key probabilistic events that Lemmas 8 and 9 depend on.

We first briefly describe the role Events (1)–(3) (Section 3.1) play in the proofs of these lemmas. Applying the Invariant after scale $k-1$ to v implies that a large number of nodes in N have degree at most $\Delta/2^{k-1}+\alpha$. This set of "low degree" nodes is large enough for us to consider Event (2) at these nodes and using Theorem 6 we see that a large fraction of these nodes have priority greater than their parents (with sufficiently high probability). Conditioning on this event, we then consider Event (1) at the "low degree" nodes in N whose priorities are larger than priorities of parents. We then apply Theorem 5 to conclude that with probability at least $1-1/\Delta^2$ at least one node in N joins the MIS, thereby eliminating v and yielding Lemma 8. To obtain Lemma 9, we repeatedly consider Event (3) at the nodes in N and apply Theorem 7 to obtain a decay of roughly $1/\alpha^8$ fraction, after each iteration with sufficiently high probability. Performing $\Lambda = \Theta(\alpha^8(\log\alpha + \log\log\Delta)$ iterations is enough to reduce $|N|$ to at most $\Delta/2^{k+2}$ with sufficiently high probability. Lemmas 8 and 9 immediately lead to Theorem 10.

▶ **Lemma 8.** *If $|N| > 130\alpha^4 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$ at the beginning of Scale k, then v is eliminated with probability at least $1 - 1/\Delta^2$ after the first iteration in Scale k.*

Proof. We focus on the first iteration of Scale k. By applying the INVARIANT at the end of Scale $k-1$ to node v, we see that v has at most $\Delta/2^{k+1}$ neighbors with degree more than $\Delta/2^{k-1} + \alpha$. Thus among the nodes in N, there are at least

$$|N| - \Delta/2^{k+1} > 130\alpha^4 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1} - \Delta/2^{k+1} > 129\alpha^4 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$$

with degree at most $\Delta/2^{k-1} + \alpha$. Let $N_{low} \subseteq N$ denote the subset of N of nodes with degree at most $\Delta/2^{k-1} + \alpha$. (Thus, we have just established that $|N_{low}| > 129\alpha^4 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$.) Since N_{low} is large enough, we can apply Theorem 6 to conclude that with probability at least $1 - 1/\Delta^4$, at least $|N_{low}|/2\alpha$ nodes in N_{low} have priorities that are larger than priorities of their parents. (Recall that this is Event (2) at N_{low}.) Call this event E_{par} and let $N_{par} \subseteq N_{low}$ denote the subset of nodes in N_{low} whose priorities are larger than priorities of their parents. Thus, if we condition on E_{par}, we get that $|N_{par}| \geq |N_{low}|/2\alpha$. We now apply Theorem 5 to the set N_{par} to get a lower bound on the probability that N_{par} contains a node whose priority is greater than the priorities of all children. Letting F denote this event, we get the lower bound:

$$Pr(F) \geq 1 - \left(1 - \frac{1}{\Delta(N_{par})}\right)^{|N_{par}|/(2\alpha^2)}.$$

Since $N_{par} \subseteq N_{low}$, we know that $\Delta(N_{par}) \leq \Delta/2^{k-1} + \alpha$ and if we condition on E_{par}, we know that $|N_{par}| > |N_{low}|/2\alpha > 64\alpha^3 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$.

$$Pr(F|E_{par}) \geq 1 - \left(1 - \frac{1}{\Delta/2^{k-1} + \alpha}\right)^{\frac{|N_{par}|}{2\alpha^2}} \geq 1 - \exp\left(-\frac{2^{k-1}}{2\Delta\alpha} \cdot 32\alpha \ln^2 \Delta \cdot \frac{\Delta}{2^{k+1}}\right) \geq 1 - 1/\Delta^4.$$

Since E_{par} occurs with probability at least $1 - 1/\Delta^4$, we conclude that

$$Pr(F) = Pr(E|E_{par}) \cdot Pr(E_{par}) \geq (1 - 1/\Delta^4)(1 - 1/\Delta^4) \geq (1 - 1/\Delta^2). \quad \blacktriangleleft$$

▶ **Lemma 9.** *If $|N| \leq 130\alpha^4 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$ at the beginning of Scale k, then after the first Λ/p iterations of Scale k, $|N| \leq \Delta/2^{k+2}$ with probability at least $1 - 1/\Delta^2$.*

Proof. Let n_i denote the size of N before iteration i, $1 \leq i \leq \Lambda/p$, in Scale k and let $n_{\Lambda/p+1}$ denote the size of N after the $\Lambda/p + 1$ iteration in Scale k. Suppose that $n_{\Lambda/p+1} > \Delta/2^{k+2}$. Then, $n_i > \Delta/2^{k+2}$ for all i, $1 \leq i \leq \Lambda$ and so we can appeal to Theorem 7 and conclude that for all $i \in [\Lambda/p]$

$$\Pr\left(n_{i+1} \leq \left(1 - \frac{1}{8\alpha^2(32\alpha^6 + 1)}\right) \cdot n_i\right) \geq 1 - 1/\Delta^3.$$

By the union bound,

$$Pr\left(\text{There exists } i : n_{i+1} > \left(1 - \frac{1}{8\alpha^2(32\alpha^6 + 1)}\right)\right) \leq \frac{\Lambda/p}{\Delta^3} \leq \frac{1}{\Delta^2}.$$

In other words, with probability at least $1 - 1/\Delta^2$, $n_{i+1} \leq (1 - 1/(2(32\alpha^6 + 1))) \cdot n_i$ for Λ/p iterations. Therefore, with probability at least $1 - 1/\Delta^2$,

$$n_{\Lambda+1} \leq \left(1 - \frac{1}{8\alpha^2(32\alpha^6 + 1)}\right)^{\Lambda/p} \cdot n_1.$$

We now observe that

$$\left(1 - \frac{1}{8\alpha^2(32\alpha^6+1)}\right)^{\Lambda/p} \leq \exp\left(-\frac{1}{8\alpha^2(32\alpha^6+1)} \cdot \frac{\Lambda}{p}\right).$$

This implies that choosing Λ at least $p \cdot 8\alpha^2(32\alpha^6+1) \cdot \ln(260\alpha^4 \ln^2 \Delta)$ suffices to guarantee that

$$\left(1 - \frac{1}{8\alpha^2(32\alpha^6+1)}\right)^{\Lambda/p} \cdot n_1 \leq \frac{n_1}{260\alpha^4 \ln^2 \Delta}.$$

Now note that we choose $\Lambda = \lceil p \cdot 8\alpha^2(32\alpha^6+1) \cdot \ln(260\alpha^4 \ln^2 \Delta) \rceil$ in Algorithm BoundedArbIndependentSet. Since $n_1 \leq 130\alpha^4 \cdot \ln^2 \Delta \cdot \Delta/2^{k+1}$, it follows that with probability at least $1 - 1/\Delta^2$, $|N| \leq \Delta/2^{k+2}$ after Λ/p iterations of scale k. ◀

▶ **Theorem 10.** *In any Scale k, a node v that is in V_{IB} at the start of the Scale is included in B with probability at most $1/\Delta^{2p}$, independent of random choices of nodes outside its three neighborhood.*

Proof. We view the Λ iterations of each scale in *chunks* of Λ/p consecutive iterations. As a direct consequence of Lemmas 8 and 9, a node v violates the invariant with probability at most $1/\Delta^2$ after a chunk. The probability that a node is bad at the end of the scale, is equal to the probability that its bad at the end of each chunk. Thus, the probability that a node is bad at the end of the Scale is at most $(1/\Delta^2)^p = 1/\Delta^{2p}$.

We now argue that the event that v joins B after Scale k is bounded independently of nodes outside v's three-neighborhood. A node v joins B if it violates the invariant at the end of a scale. This means that many neighbors of v in N survive the scale. The survival of these high degree nodes depends on their neighbors (v's two-neighborhood) not joining the MIS. The event that nodes in v's two-neighborhood do not join the MIS, in turn, depends on these nodes choosing higher priorities than their neighbors, which can be at most three hops away from v. Thus v joins the bad set with probability at most $1/\Delta^{2p}$, independent of random choices made by nodes outside v's three-neighborhood. ◀

3.3 Finishing Up the MIS Computation

Theorem 10 shows that every node joins B with probability at most $1/\Delta^{2p}$. For $p \geq 9$, this has the following immediate consequence, shown in [3, 2].

▶ **Lemma 11.** *All connected components in the subgraph induced by B have at most $(\Delta^6 \cdot c \log_\Delta n)$ nodes with probability at least $1 - n^{-c}$*

We now describe and analyze an algorithm, we call ArbMIS that takes the output of BoundedArbIndependentSet (Section 2) and completes the computation of an MIS. Recall that after the termination of BoundedArbIndependentSet, we get three (disjoint) sets of nodes, V_{IB}, I, and B with the following properties:

- **(i)** I is an independent set.
- **(ii)** No node in V_{IB} has more than $1176 \cdot 4\alpha^{10} \ln^2 \Delta$ neighbors with degree more than $1176 \cdot 16\alpha^{10} \ln^2 \Delta + \alpha$. This follows from applying the Invariant at the end of Scale $\Theta = \left\lfloor \log\left(\frac{\Delta}{1176 \cdot 16\alpha^{10} \ln^2 \Delta}\right) \right\rfloor$.
- **(iii)** All connected components in the graph induced by B, have size less than $O(\Delta^6 \cdot c \log_\Delta n)$ with probability $1 - n^{-c}$. This follows from Lemma 11.

After BoundedArbIndependentSet has completed execution, we divide V_{IB} into the sets $V_{lo} = \{v \in V_{IB} | \deg_{IB}(v) \leq 1176 \cdot 16\alpha^{10} \ln^2 \Delta + \alpha\}$ and $V_{hi} = \{v \in V_{IB} | \deg_{IB}(v) > 1176 \cdot 16\alpha^{10} \ln^2 \Delta + \alpha\}$. By definition, $G[V_{lo}]$ has maximum degree $1176 \cdot 16\alpha^{10} \ln^2 \Delta + \alpha$ and by Property (ii) above, $G[V_{hi}]$ has maximum degree $1176 \cdot 4\alpha^{10} \ln^2 \Delta$. There are various options for computing an MIS on a arboricity-α graph with bounded degree. We use an algorithm from Barenboim et al. (Theorem 7.4, [3]) to compute an MIS of $G[V_{lo}]$ in $O(\log\log n \cdot \log \alpha + (\log\log \Delta)^2 + \alpha^2)$ time in the $\mathcal{CONGEST}$ model. Subsequently, we use the same algorithm on $V_{hi} \setminus \Gamma(I_{lo})$ to get an independent set I_{hi} in the same time. The following simple lemma (whose proof appears in the full paper [12]) shows that an MIS can be computed efficiently for each connected component in the graph induced by B.

▶ **Lemma 12.** *For each connected component in B, an MIS can be computed in time at most $O(\log \Delta + \log\log n + \alpha \log^* n)$ time, using messages of size at most $O(\log n)$.*

The total run-time of this algorithm is $O\left(\alpha^8 \cdot \log(\alpha \log \Delta) \cdot \log \Delta + \log\log n \cdot \log \alpha\right)$. If $\Delta > \alpha \cdot 2^{\sqrt{\log n / \log\log n}}$, use the independent set algorithm from [3, 2] to reduce the maximum degree to $\alpha \cdot 2^{\sqrt{\log n / \log\log n}}$ in $O(\sqrt{\log n \cdot \log\log n})$ time, using messages of size $O(\log n)$. Then, apply ArbMIS to compute an MIS for a total run time of $O(\alpha^9 \cdot \sqrt{\log n \cdot \log\log n})$. We note that this entire algorithm uses messages of size at most $O(\log n)$ i.e., it runs in the $\mathcal{CONGEST}$ model.

References

1. Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7(4):567–583, 1986.
2. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *FOCS*, pages 321–330, 2012. doi:10.1109/FOCS.2012.60.
3. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *CoRR*, abs/1202.1983, 2015. URL: http://arxiv.org/abs/1202.1983.
4. Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.
5. Dmitry Gavinsky, Shachar Lovett, Michael Saks, and Srikanth Srinivasan. A tail bound for read-k families of functions. *Random Structures & Algorithms*, 47:99–108, 2015. doi:10.1002/rsa.20532.
6. Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *SODA*, pages 270–277, 2016. doi:10.1137/1.9781611974331.ch20.
7. Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, 1986. doi:10.1016/0020-0190(86)90144-4.
8. Christoph Lenzen and Roger Wattenhofer. MIS on trees. In *PODC*, pages 41–48, 2011. doi:10.1145/1993806.1993813.
9. M. Luby. A simple parallel algorithm for the maximal independent set. *SIAM Journal on Computing*, 15:1036–1053, 1986.
10. Y. Métivier, J.M. Robson, N. Saheb-Djahromi, and A. Zemmari. An optimal bit complexity randomised distributed MIS algorithm. In *SIROCCO*, pages 323–337, 2009.
11. Sriram V. Pemmaraju and Talal Riaz. Brief announcement: Using read-k inequalities to analyze a distributed MIS algorithm. In *PODC*, pages 483–485, 2016.
12. Sriram V. Pemmaraju and Talal Riaz. Using read-k inequalities to analyze a distributed MIS algorithm. *CoRR*, abs/1605.06486, 2016.

13 Johannes Schneider and Roger Wattenhofer. A log-star distributed maximal independent set algorithm for growth-bounded graphs. In *PODC*, pages 35–44, 2008.

14 Alistair Sinclair. Randomness and computation, lecture 13. `http://www.cs.berkeley.edu/~sinclair/cs271/n13.pdf`. Accessed: 2015-08-28.

Self-Stabilizing Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance in Polynomial Steps*

Stéphane Devismes[1], David Ilcinkas[2], and Colette Johnen[3]

1 **Université Grenoble Alpes, Grenoble, France**
`stephane.devismes@imag.fr`
2 **Université Bordeaux & CNRS, LaBRI, Talence, France**
`david.ilcinkas@labri.fr`
3 **Université Bordeaux & CNRS, LaBRI, Talence, France**
`johnen@labri.fr`

Abstract

We deal with the problem of maintaining a shortest-path tree rooted at some process r in a network that may be disconnected after topological changes. The goal is then to maintain a shortest-path tree rooted at r in its connected component, V_r, and make all processes of other components detecting that r is not part of their connected component. We propose, in the composite atomicity model, a silent self-stabilizing algorithm for this problem working in semi-anonymous networks under the distributed unfair daemon (the most general daemon) without requiring any *a priori* knowledge about global parameters of the network. This is the first algorithm for this problem that is proven to achieve a polynomial stabilization time in steps. Namely, we exhibit a bound in $O(\mathtt{W}_{\max} n_{\mathtt{maxCC}}{}^3 n)$, where $\mathtt{W}_{\max}$ is the maximum weight of an edge, $n_{\mathtt{maxCC}}$ is the maximum number of non-root processes in a connected component, and n is the number of processes. The stabilization time in rounds is at most $3n_{\mathtt{maxCC}} + D$, where D is the hop-diameter of V_r.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases distributed algorithm, self-stabilization, routing algorithm, shortest path, disconnected network, shortest-path tree

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.10

1 Introduction

Given a connected undirected edge-weighted graph G, a *shortest-path (spanning) tree rooted at node r* is a spanning tree T of G, such that for every node u, the unique path from u to r in T is a shortest path from u to r in G. This data structure finds applications in the networking area (*n.b.*, in this context, nodes actually represent processes), since many distance-vector routing protocols, like *RIP* (*Routing Information Protocol*) and *BGP* (*Border Gateway Protocol*), are based on the construction of shortest-path trees. Indeed, such algorithms implicitly builds a shortest-path tree rooted at each destination.

From time to time, the network may be split into several connected components due to the network dynamics. In this case, routing to process r correctly operates only for the

* This study has been partially supported by the ANR projects Descartes (ANR-16-CE40-0023), Estate (ANR-16-CE25-0009), and Macaron (ANR-13-JS02-002). This study has been carried out in the frame of "the Investments for the future" Programme IdEx Bordeaux – cpu (ANR-10-IDEX-03-02).

processes of its connected component, V_r. Consequently, in other connected components, information to reach r should be removed to gain space in routing tables, and to discard messages destined to r (which are unable to reach r anyway) and thus save bandwidth. The goal is then to make the network converging to a configuration where every process of V_r knows a shortest path to r and every other process detects that r is not in its own connected component. We call this problem the *Disconnected Components Detection and rooted Shortest-Path tree Maintenance* (DCDSPM). Notice that a solution to this problem allows to prevent the well-known *count-to-infinity* problem [27], where the distances to some unreachable process keep growing in routing tables because no process is able to detect the issue.

When topological changes are infrequent, they can be considered as transient faults [30] and self-stabilization [16] – a versatile technique to withstand *any* finite number of transient faults in a distributed system – becomes an attractive approach. A self-stabilizing algorithm is able to recover without external (*e.g.*, human) intervention a correct behavior in finite time, regardless of the *arbitrary* initial configuration of the system, and therefore, also after the occurrence of transient faults, provided that these faults do not alter the code of the processes.

A particular class of self-stabilizing algorithms is that of silent algorithms. A self-stabilizing algorithm is *silent* [19] if it converges to a global state where the values of communication registers used by the algorithm remain fixed. Silent (self-stabilizing) algorithms are usually proposed to build distributed data structures, and so are well-suited for the problem considered here. As quoted in [19], the silent property usually implies more simplicity in the algorithm design, moreover a silent algorithm may utilize less communication operations and communication bandwidth.

For sake of simplicity, we consider here a single destination process r, called the *root*. However, the solution we will propose can be generalized to work with any number of destinations, provided that destinations can be distinguished. In this context, we do not require the network to be fully identified. Rather, r should be distinguished among other processes, and all non-root processes are supposed to be identical: we consider semi-anonymous networks.

In this paper, we propose a silent self-stabilizing algorithm, called Algorithm RSP, for the DCDSPM problem with a single destination process in semi-anonymous networks. Algorithm RSP does not require any *a priori* knowledge of processes about global parameters of the network, such as its size or its diameter. Algorithm RSP is written in the locally shared memory model with composite atomicity introduced by Dijkstra [16], which is the most commonly used model in self-stabilization. In this model, executions proceed in (atomic) steps, and a self-stabilizing algorithm is silent if and only if all its executions are finite. Moreover, the asynchrony of the system is captured by the notion of *daemon*. The weakest (*i.e.*, the most general) daemon is the *distributed unfair daemon*. Hence, solutions stabilizing under such an assumption is highly desirable, because it works under any other daemon assumption. Moreover, time complexity (the *stabilization time*, mainly) can be bounded in terms of steps only if the algorithm works under an unfair daemon. Otherwise (*e.g.*, under a weakly fair daemon), time complexity can only be evaluated in terms of rounds, which capture the execution time according to the slowest process.

Self-stabilizing algorithms under the distributed unfair daemon are numerous in the literature *e.g.*, [14, 12, 21, 5, 1]. However, analyses of the stabilization time in steps is rather unusual and this may be an important issue. Indeed, recently, several self-stabilizing algorithms, which work under a distributed unfair daemon, have been shown to have an exponential stabilization time in steps in the worst case. In [1], silent leader election algorithms

from [12, 14] are shown to be exponential in steps in the worst case. In [15], the Breadth-First Search (BFS) algorithm of Huang and Chen [23] is also shown to be exponential in steps. Finally, in [22] authors show that the first silent self-stabilizing algorithm for the DCDSPM problem (still assuming a single destination) they proposed in [21] is also exponential in steps. Precisely, they exhibit a family of graph of $4k + 2$ nodes on which there is an execution of their algorithm containing at least 2^{k+2} steps.

1.1 Contribution

Algorithm RSP proposed here is the first silent self-stabilizing algorithm for the DCDSPM problem which achieves a polynomial stabilization time in steps. Namely, assuming that the edge weights are positive integers, the stabilization time of RSP is at most $(\mathtt{W}_{\max} {n_{\mathtt{maxCC}}}^3 + (3 - \mathtt{W}_{\max}) n_{\mathtt{maxCC}} + 3)(n-1)$, where $\mathtt{W}_{\max}$ is the maximum weight of an edge, $n_{\mathtt{maxCC}}$ is the maximum number of non-root processes in a connected component, and n is the number of processes. (*N.b.*, this stabilization time is less than or equal to $\mathtt{W}_{\max} n^4$, for all $n \geq 3$.) Moreover, when all weights are equal to one, the problem reduces to a BFS tree maintenance and the step complexity becomes at most $({n_{\mathtt{maxCC}}}^3 + 2n_{\mathtt{maxCC}} + 3)(n-1)$, which is less than or equal to n^4 for all $n \geq 2$. We also studied the stabilization time in rounds of our solution. We established a bound of at most $3n_{\mathtt{maxCC}} + D$ rounds, where D is the hop-diameter of V_r (defined as the maximum over all pairs $\{u, v\}$ of nodes in V_r of the minimum number of edges in a shortest path from u to v).

1.2 Related Work

To the best of our knowledge, only one self-stabilizing algorithm for the DCDSPM problem has been previously proposed in the literature [21]. This algorithm is silent and works under the distributed unfair daemon, but, as previously mentioned, it is exponential in steps. However, it assumes positive *real* weights whereas Algorithm RSP assumes positive *integer* weights[1], and it has a slightly better stabilization time in rounds, precisely at most $2(n_{\mathtt{maxCC}} + 1) + D$ rounds[2].

There are several shortest-path spanning tree algorithms in the literature that do not consider the problem of disconnected components detection. The oldest distributed algorithms are inspired by the Bellman-Ford algorithm [3, 20]. Self-stabilizing shortest-path spanning tree algorithms have then being proposed in [6, 25], but these two algorithms are proven assuming a central daemon, which only allows sequential executions. However, in [24], Tetz Huang proves that these algorithms actually work assuming the distributed unfair daemon. Nevertheless, no upper bounds on the stabilization time (in rounds or steps) are given.

Self-stabilizing shortest-path spanning tree algorithms are also given in [2, 9, 26]. These algorithms additionally ensure the loop-free property in the sense that they guarantee that a spanning tree structure is always preserved while edge costs change dynamically. Now, none of these papers consider the unfair daemon, and consequently their step complexity cannot be analyzed.

Whenever all edges have weight one, shortest-path trees correspond to BFS trees. In [13], the authors introduce the *disjunction* problem as follows. Each process has a constant input

[1] It is not difficult to see that our algorithm is in fact correct even when weights are positive reals. However, our bound on the number of steps is valid only for integer weights, not for arbitrary real ones.

[2] In fact, [21] announced $2n + D$ rounds, but it is easy to see that this complexity can be reduced to $2(n_{\mathtt{maxCC}} + 1) + D$.

bit, 0 or 1. Then, the problem consists for each process in computing an output value equal to the disjunction of all input bits in the network. Moreover, each process with input bit 1 (if any) should be the root of a tree, and each other process should join the tree of the closest input bit 1 process, if any. If there is no process with input bit 1, the execution should terminate and all processes should output 0. The proposed algorithm is silent and self-stabilizing. Hence, if we set the input of a process to 1 if and only if it is the root, then their algorithm solves the DCDSPM problem when all edge-weights are equal to one, since any process which is not in V_r will compute an output 0, instead of 1 for the processes in V_r. The authors show that their algorithm stabilizes in $O(n)$ rounds, but no step complexity analysis given. Now, as their approach is similar to [14], it is not difficult to see that their algorithm is also exponential in steps.

Several other self-stabilizing BFS tree algorithms have been proposed, but without considering the problem of disconnected components detection. Chen *et al.* present the first self-stabilizing BFS tree construction in [8] under the central daemon. Huang and Chen present the first self-stabilizing BFS tree construction in [23] under the distributed unfair daemon, but recall that this algorithm has been proven to be exponential in steps in [15]. Finally, notice that these two latter algorithms [23, 15] require that processes know the exact number of processes in the network.

According to our knowledge, only the following works [10, 11] take interest in the computation of the number of steps required by their BFS algorithms. The algorithm in [10] is not silent and has a stabilization time in $O(\Delta n^3)$ steps, where Δ is the maximum degree in the network. The silent algorithm given in [11] has a stabilization time $O(D^2)$ rounds and $O(n^6)$ steps.

1.3 Roadmap

In the next section, we present the computational model and basic definitions. In Section 3, we describe Algorithm RSP. Its proof of correctness and a complexity analysis in steps are given in Section 4. Finally, an analysis of the stabilization time in rounds is proposed in Section 5.

2 Preliminaries

We consider *distributed systems* made of $n \geq 1$ interconnected processes. Each process can directly communicate with a subset of other processes, called its *neighbors*. Communication is assumed to be bidirectional. Hence, the topology of the system can be represented as a simple undirected graph $G = (V, E)$, where V is the set of processes and E the set of edges, representing communication links. Every process v can distinguish its neighbors using a *local labeling* of a given datatype *Lbl*. All labels of v's neighbors are stored into the set $\Gamma(v)$. Moreover, we assume that each process v can identify its local label in the set $\Gamma(u)$ of each neighbor u. Such labeling is called *indirect naming* in the literature [29]. By an abuse of notation, we use v to designate both the process v itself, and its local labels.

Each edge $\{u, v\}$ has a strictly positive Integer *weight*, denoted by $\omega(u, v)$. This notion naturally extends to paths: the weight of a path in G is the sum of its edge weights. The weighted distance between the processes u and v, denoted by $d(u, v)$, is the minimum weight of a path from u to v. Of course, $d(u, v) = \infty$ if and only if u and v belong to two distinct connected components of G.

We use the *composite atomicity model of computation* [16, 18] in which the processes communicate using a finite number of locally shared registers, called *variables*. Each process

can read its own variables and those of its neighbors, but can write only to its own variables. The *state* of a process is defined by the values of its local variables. The union of states of all processes determines the *configuration* of the system.

A *distributed algorithm* consists of one local program per process. We consider semi-uniform algorithms, meaning that all processes except one, the *root* r, execute the same program. In the following, for every process u, we denote by V_u the set of nodes (including u) in the same connected component of G as u. In the following V_u is simply referred to as the connected component of u. We denote by $n_{\texttt{maxCC}}$ the maximum number of non-root processes in a connected component of G. By definition, $n_{\texttt{maxCC}} \leq n-1$.

The *program* of each process consists of a finite set of *rules* of the form $label : guard \rightarrow action$. *Labels* are only used to identify rules in the reasoning. A *guard* is a Boolean predicate involving the state of the process and that of its neighbors. The *action* part of a rule updates the state of the process. A rule can be executed only if its guard evaluates to *true*; in this case, the rule is said to be *enabled*. A process is said to be enabled if at least one of its rules is enabled. We denote by $Enabled(\gamma)$ the subset of processes that are enabled in configuration γ.

When the configuration is γ and $Enabled(\gamma) \neq \emptyset$, a *daemon* selects a non-empty set $\mathcal{X} \subseteq Enabled(\gamma)$; then every process of $\mathcal{X}$ *atomically* executes one of its enabled rules, leading to a new configuration γ', and so on. The transition from γ to γ' is called a *step*. The possible steps induce a binary relation over $\mathcal{C}$, denoted by $\mapsto$. An *execution* is a maximal sequence of configurations $e = \gamma_0\gamma_1 \ldots \gamma_i \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no rule is enabled at any process.

As previously stated, each step from a configuration to another is driven by a daemon. In this paper we assume the daemon is *distributed* and *unfair*. "Distributed" means that while the configuration is not terminal, the daemon should select at least one enabled process, maybe more. "Unfair" means that there is no fairness constraint, *i.e.*, the daemon might never select an enabled process unless it is the only enabled process.

In the composite atomicity model, an algorithm is *silent* if and only if all its executions are finite. Hence, we can define silent self-stabilization as follows.

▶ **Definition 1** (Silent Self-Stabilization)**.** Let $\mathcal{L}$ be a non-empty subset of configurations, called set of legitimate configurations. A distributed system is silent and self-stabilizing under the daemon S for $\mathcal{L}$ if and only if the following two conditions hold:

- all executions under S are finite, and
- all terminal configurations belong to $\mathcal{L}$.

We use the notion of *round* [17] to measure the time complexity. The first round of an execution $e = \gamma_0, \gamma_1, \cdots$ is the minimal prefix $e_1 = \gamma_0, \cdots, \gamma_j$, such that every process that is enabled in γ_0 either executes a rule or is neutralized during a computation step of e_1. A process v is *neutralized* during a computation step $\gamma_i \mapsto \gamma_{i+1}$, if v is enabled in γ_i but not in configuration γ_{i+1}. Let e' be the suffix $\gamma_j, \gamma_{j+1}, \cdots$ of e. The second round of e is the first round of e', and so on.

The *stabilization time* of a silent self-stabilizing algorithm is the maximum time (in steps or rounds) over every execution (starting from any initial configuration) to reach a terminal (legitimate) configuration.

3 Algorithm RSP

This section is devoted to the presentation of our algorithm, Algorithm RSP (which stands for *Rooted Shortest-Path*). The code of Algorithm RSP is given in Algorithms 1 and 2.

3.1 Variables

In RSP, each process u maintains three variables: st_u, par_u, and d_u. Those three variables are constant for the root process[3], r: $st_r = C$, $par_r = \perp$[4], and $d_r = 0$. For each non-root process u, we have:

- $st_u \in \{I, C, EB, EF\}$, this variable gives the *status* of the process. I, C, EB, and EF respectively stand for *Isolated*, *Correct*, *Error Broadcast*, and *Error Feedback*. The two first states, I and C, are involved in the normal behavior of the algorithm, while the two last ones, EB and EF, are used during the correction mechanism. Precisely, $st_u = C$ (resp. $st_u = I$) means that u believes it is in V_r (resp. not in V_r). The meaning of status EB and EF will be further detailed in Subsection 3.3.
- $par_u \in Lbl$, a *parent pointer*. If $u \in V_r$, par_u should designate a neighbor of u, referred to as its *parent*, and in a terminal configuration, the parent pointers exhibit a shortest path from u to r.
  Otherwise ($u \notin V_r$), the variable is meaningless.
- $d_u \in \mathbb{N}^*$, the *distance* value. If $u \in V_r$, then in a terminal configuration, d_u gives the weight of the shortest path from u to r.
  Otherwise ($u \notin V_r$), the variable is meaningless.

3.2 Normal Execution

Consider any configuration, where every process $u \neq r$ satisfies $st_u = I$, and refer to such a configuration as a *normal initial configuration*. Each configuration reachable from a *normal initial configuration* is called a *normal configuration*, otherwise it is an *abnormal configuration*. Recall that $st_r = C$ in all configurations. Then, starting from a normal initial configuration, all processes in a connected component different from V_r is disabled forever. Focus now on the connected component V_r. Each neighbor u of r is enabled to execute $\mathbf{R_R}(u)$. A process eventually chooses r as parent by executing this rule, which in particular sets its status to C. Then, executions of rule $\mathbf{R_R}$ are asynchronously propagated in V_r until all its processes have status C: when a process u with status I finds one of its neighbor with status C it executes $\mathbf{R_R}(u)$: u takes status C and chooses as parent its neighbor v with status C such that $d_v + \omega(u, v)$ is minimum, d_u being updated accordingly. In parallel, rules $\mathbf{R_C}$ are executed to reduce the weight of the tree rooted at r: when a process u with status C can reduce d_u by selecting another neighbor with status C as parent, it chooses the one allowing to minimize d_u by executing $\mathbf{R_C}(u)$. Hence, eventually, the system reaches a terminal configuration, where the tree rooted at r is a shortest-path tree spanning all processes of V_r.

3.3 Error Correction

Assume now that the system is in an abnormal configuration. Some non-root processes locally detect that their state is inconsistent with that of their neighbors. We call *abnormal roots* such processes. Informally (see Subsection 3.4 for the formal definition), a process $u \neq r$ is an *abnormal root* if u is not isolated (*i.e.*, $st_u \neq I$) and satisfies one of the following four conditions:

1. its parent pointer does not designate a neighbor,
2. its parent has status I,

[3] We should emphasize that the use of constants at the root is not a limitation, rather it allows to simplify the design and proof of the algorithm. Indeed, these constants can be removed by adding a rule to correct all root's variables, if necessary, within a single step.

[4] $\perp$ is a particular value which is different from any value in Lbl.

3. its distance value d_u is inconsistent with the distance value of its parent, or
4. its status is inconsistent with the status of its parent.

Every non-root process u that is not an abnormal root satisfies one of the two following cases. Either u is *isolated*, *i.e.*, $st_u = I$, or u points to some neighbors (*i.e.*, $par_u \in \Gamma(u)$) and the state of u is coherent *w.r.t.* the state of its parent. In this latter case, $u \in children(par_u)$, *i.e.*, u is a "real" child of its parent (see Subsection 3.4 for the formal definition). Notice that every so-called *parent path* $\mathcal{P} = u_1, \ldots, u_k$ such that $\forall i, 0 \leq i < k, u_i \in children(u_{i+1})$ is acyclic, and if $\mathcal{P}$ is maximal, then u_k is either r, or an abnormal root. Hence, we define the normal tree $T(r)$ (resp. an abnormal tree $T(v)$, for any abnormal root v) as the set of all processes u such that there is a parent path from u to r (resp. v).

Then, the goal is to remove all abnormal trees so that the system recovers a normal configuration. We remove these abnormal trees in a top-down manner starting from their roots. Now, we have to prevent the following situation: an abnormal root leaves its tree and later selects as parent a process that was previously in its tree. Hence, the idea is to freeze each abnormal tree, before removing it. By freezing we mean assigning each member of the tree to a particular state, here EF, so that (1) no member u of the tree is allowed to execute $\mathbf{R_R}(u)$, and (2) no process v can join the tree by executing $\mathbf{R_C}(v)$. Once frozen, the tree can be safely deleted from its root to its leaves.

The freezing mechanism (inspired from [4]) is achieved using the status EB and EF. If a process is not involved into any freezing operation, then its status is I or C. Otherwise, it has status EB or EF and no neighbor can select it as its parent. These two latter states are actually used to perform a "Propagation of Information with Feedback" [7, 28] in the abnormal trees. This is why status EB means "Error Broadcast" and EF means "Error Feedback". From an abnormal root, the status EB is broadcast down in the tree. Then, once the EB wave reaches a leaf, the leaf initiates a convergecast EF-wave. Once the EF-wave reaches the abnormal root, the tree is said to be *dead*, meaning that all processes in the tree have status EF and, consequently, no other process can join it. So, the tree can be safely deleted from its abnormal root toward its leaves. There is two possibilities for the deletion. If the process u to be deleted has a neighbor with status C, then it executes rule $\mathbf{R_R}(u)$ to directly join another "alive" tree. Otherwise, u becomes isolated by executing rule $\mathbf{R_I}(u)$, and u may join another tree later.

Let u be a process belonging to an abnormal tree at which it is not the root. Let v be its parent. From the previous explanation, it follows that during the correction, $(st_v, st_u) \in \{(C,C),(EB,C),(EB,EB),(EB,EF),(EF,EF)\}$ until v resets by $\mathbf{R_R}(v)$ or $\mathbf{R_I}(v)$. Now, due to the arbitrary initialization, the status of u and v may not be coherent, in this case u should also be an abnormal root. Precisely, as formally defined below, the status of u is incoherent *w.r.t* the status of its parent v if $st_u \neq st_v$ and $st_v \neq EB$.

Actually, the freezing mechanism insures that if a process is the root of an abnormal alive tree, it is in that situation since the initial configuration (see Lemma 15, page 10:10). The polynomial step complexity mainly relies on this strong property.

3.4 Definitions

▶ **Definition 2** (Abnormal Root). For every process $u \neq r$, $abRoot(u) \equiv st_u \neq I \wedge \big[par_u \notin \Gamma(u) \vee st_{par_u} = I \vee d_u < d_{par_u} + \omega(u, par_u) \vee (st_u \neq st_{par_u} \wedge st_{par_u} \neq EB)\big]$.

Every process $u \neq r$ that satisfies $abRoot(u)$ is said to be an *abnormal root*.

▶ **Definition 3** (Alive Abnormal Root). A process $u \neq r$ is said to be an *alive abnormal root* (resp. a *dead abnormal root*) if u is an abnormal root and has a status different from EF

Algorithm 1: Code of RSP for the root process r.

Constants:

$st_r = C$
$par_r = \perp$
$d_r = 0$

Algorithm 2: Code of RSP for any process $u \neq r$.

Variables:

$st_u \in \{I, C, EB, EF\}$
$par_u \in Lbl$
$d_u \in \mathbb{N}^*$

Predicates:

$P\_reset(u) \equiv st_u = EF \wedge abRoot(u)$
$P\_correction(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \wedge d_v + \omega(u,v) < d_u)$

Macro:

$computePath(u)$: $par_u := \text{argmin}_{(v \in \Gamma(u) \wedge st_v = C)}(d_v + \omega(u,v))$;
$d_u := d_{par_u} + \omega(u, par_u)$;
$st_u := C$

Rules

$\mathbf{R_C}(u)$	:	$st_u = C \wedge P\_correction(u)$	$\rightarrow$	$computePath(u)$
$\mathbf{R_{EB}}(u)$	:	$st_u = C \wedge \neg P\_correction(u) \wedge (abRoot(u) \vee st_{par_u} = EB)$	$\rightarrow$	$st_u := EB$
$\mathbf{R_{EF}}(u)$	:	$st_u = EB \wedge (\forall v \in children(u) \mid st_v = EF)$	$\rightarrow$	$st_u := EF$
$\mathbf{R_I}(u)$	:	$P\_reset(u) \wedge (\forall v \in \Gamma(u) \mid st_v \neq C)$	$\rightarrow$	$st_u := I$
$\mathbf{R_R}(u)$	:	$(P\_reset(u) \vee st_u = I) \wedge (\exists v \in \Gamma(u) \mid st_v = C)$	$\rightarrow$	$computePath(u)$

(resp. has status EF).

▶ **Definition 4** (Children)**.** For every process v, $children(v) = \{u \in \Gamma(v) \mid st_v \neq I \wedge st_u \neq I \wedge par_u = v \wedge d_u \geq d_v + \omega(u,v) \wedge (st_u = st_v \vee st_v = EB)\}$.

▶ **Definition 5** (Branch)**.** A *branch* is a maximal sequence of processes $v_1, \cdots, v_k$ for some integer $k \geq 1$, such that v_1 is r or an abnormal root and, for every $1 \leq i < k$, we have $v_{i+1} \in children(v_i)$. The process v_i is said to be at *depth* i and $v_i, \cdots, v_k$ is called a *sub-branch*. If $v_1 \neq r$, the branch is said to be *illegal*, otherwise, the branch is said to be *legal*.

▶ **Observation 6.** *A branch depth is at most n. A process v having status I does not belong to any branch. If a process v has status C (resp. EF), then all processes of a sub-branch starting at v have status C (resp. EF).*

4 Correctness and Step Complexity of Algorithm RSP

4.1 Partial Correctness

Before proceeding with the proof of correctness and the step complexity analysis, we define some useful concepts.

▶ **Definition 7** (Legitimate State)**.** A process u is said to be in a *legitimate state* if u satisfies one of the following three conditions:

1. $u = r$,
2. $u \neq r$, $u \in V_r$, $st_u = C$, $d_u = d(u,r)$, and $d_u = d_{par_u} + \omega(u, par_u)$, or
3. $u \notin V_r$ and $st_u = I$.

▶ **Observation 8.** *Every process $u \neq r$ such that $st_u = C$ and $d_u \neq d_{par_u} + \omega(u, par_u)$ is enabled.*

▶ **Definition 9** (Legitimate Configuration)**.** A *legitimate configuration* is any configuration where every process is in a legitimate state. We denote by $\mathcal{LC}_{\mathsf{RSP}}$ the set of all legitimate configurations of Algorithm RSP.

Let γ be a configuration. Let $T_\gamma = (V_r, E_{T_\gamma})$ be the subgraph, where $E_{T_\gamma} = \{\{p, q\} \in E \mid p \in V_r \setminus \{r\} \wedge par_p = q\}$. By Definition 7 (point 2), we deduce the following observation.

▶ **Observation 10.** *In every legitimate configuration γ, T_γ is a shortest-path tree spanning all processes of V_r.*

We now prove that the set of terminal configurations is exactly the set of legitimate configurations. We start by proving the following intermediate statement.

▶ **Lemma 11.** *In any terminal configuration, every process has either status I or C.*

Proof. This is trivially true for the root process, r. Assume that there exists a non-root process with status EB in a terminal configuration γ. Consider the non-root process u with status EB having the largest distance value d_u in γ. In γ, no process v with status C can be a child of u, otherwise either $\mathrm{R_{EB}}$ or $\mathrm{R_C}$ is enabled at v in γ, a contradiction. Moreover, by maximality of d_u, u cannot have a child with status EB in γ. Therefore, in γ process u has no child or it has only children with status EF, and thus rule $\mathrm{R_{EF}}$ is enabled at u, a contradiction. Thus, every process has status C, I, or EF in γ.

Assume now that there exists a non-root process with status EF in a terminal configuration γ. Consider the process u with status EF having the smallest distance value d_u in γ. By construction, u is an abnormal root in γ. So, either $\mathrm{R_I}$ or $\mathrm{R_R}$ is enabled at u in γ, a contradiction. ◀

The next lemma, Lemma 12, deals with the connected components that do not contain r, if any. Then, Lemma 13 deals with the connected component V_r.

▶ **Lemma 12.** *In any terminal configuration, every process that does not belong to V_r is in a legitimate state.*

Proof. Consider, by contradiction, that there exists a process u that belongs to the connected component CC other than V_r which is not in a legitimate state in some terminal configuration γ. By definition, u is not the root, moreover it has status C in γ, by Lemma 11. Without loss of generality, assume that u is the process of CC with status C having the smallest distance value d_u in γ. By construction, u is an abnormal root in γ. Thus, rule $\mathrm{R_{EB}}$ is enabled at u in γ, a contradiction. ◀

▶ **Lemma 13.** *In any terminal configuration, every process of V_r is in a legitimate state.*

Proof. Assume, by contradiction, that there exists a terminal configuration γ where at least one process in the connected component V_r is not in a legitimate state.

Assume also that there exists some process of V_r that has status I in γ. Consider now a process u of V_r such that in γ, u has status I and at least one of its neighbors has status C. Such a process exists because no process has status EB or EF in γ (Lemma 11), but at least one process of V_r has status C, namely r. Obviously, $\mathrm{R_R}$ is enabled at u in γ, a contradiction. So, every process in V_r must have status C in γ. Moreover, for all processes in V_r, we have $d_u = d_{par_u} + \omega(par_u, u)$ in γ, otherwise $\mathrm{R_C}$ is enabled at some process of V_r in γ.

Assume now that there exists a process u such that $d_u < d(u,r)$ in γ. Consider a process u of V_r having the smallest distance value d_u among the processes in V_r such that $d_u < d(u,r)$ in γ. By definition, $u \neq r$ and we have $d_u > d_{par_u}$ in γ, so $d_{par_u} \geq d(par_u, r)$ in γ. Hence, we can conclude that $d_u \geq d(u,r)$ in γ, a contradiction. So, every process u in V_r satisfies $d_u \geq d(u,r)$ in γ.

Finally, assume that there exists a process u such that $d_u > d(u,r)$ in γ. Consider a process u in V_r having the smallest distance to r among the processes in V_r such that $d_u > d(u,r)$ in γ. By definition, $u \neq r$ and there exists some process v in $\Gamma(u)$ such that $d(u,r) = d(v,r) + \omega(u,v)$ in γ. Thus, we have $d_v = d(v,r)$ in γ. So, $\mathrm{R_C}$ is enabled at u in γ, a contradiction. ◀

After noticing that any legitimate configuration is a terminal one (by construction of the algorithm), we deduce the following corollary from the two previous lemmas.

▶ **Corollary 14.** *For every configuration γ, γ is terminal if and only if γ is legitimate.*

4.2 Termination

In this section, we establish that every execution of Algorithm RSP under a distributed unfair daemon is finite. More precisely, we compute the following bound on the number of steps of every execution: $[\mathtt{W}_{\max} {n_{\mathtt{maxCC}}}^3 + (3 - \mathtt{W}_{\max}) n_{\mathtt{maxCC}} + 3](n-1)$, where n is the number of processes, $\mathtt{W}_{\max}$ is the maximum weight of an edge, and $n_{\mathtt{maxCC}}$ is the maximum number of non-root processes in a connected component.

▶ **Lemma 15.** *No alive abnormal root is created along any execution.*

Proof. Let $\gamma \mapsto \gamma'$ be a step. Let u be a non-root process that is not an *alive abnormal root* in γ, and let v be the process such that $par_u = v$ in γ'. If the status of u is EF or I in γ', then u is not an alive abnormal root in γ'. Consider then the case where u has status EB in γ'. The only rule u can execute in $\gamma \mapsto \gamma'$ is $\mathrm{R_{EB}}$. Whether u executes $\mathrm{R_{EB}}$ or not, par_u is also v in γ. Since u is not an alive abnormal root in γ, and thus not an abnormal root either, v is not r and necessarily has status EB in γ in either case. Moreover, u belongs to $children(v)$ in γ. So, v is not enabled in γ and $u \in children(v)$ remains true in γ'. Hence, we can conclude that u is still not an alive abnormal root in γ'.

Let study the other case, *i.e.*, u has status C in γ'. During $\gamma \mapsto \gamma'$, the only rules that u may execute are $\mathrm{R_R}$ or $\mathrm{R_C}$. So, we have $st_v = C$ in γ, because it is a requirement to execute one of the two previous rules, or $parent_u = v$ in γ. During $\gamma \mapsto \gamma'$, the only rules that v may execute are $\mathrm{R_C}$ or $\mathrm{R_{EB}}$. Thus, during $\gamma \mapsto \gamma'$, v either takes the status EB, decreases its distance value, or does not change the value of its variables. In either cases, u belongs to $children(v)$ in γ', which prevents u from being an alive abnormal root in γ'. ◀

Let $AAR(\gamma)$ be the set of alive abnormal roots in any configuration γ. From the previous lemma, we know that, for every step $\gamma \mapsto \gamma'$, we have $AAR(\gamma') \subseteq AAR(\gamma)$. So, we can use the notion of *u-segment* (inspired from [1]) to bound the total number of steps in an execution.

▶ **Definition 16** (u-Segment)**.** Let u be any non-root process. Let $e = \gamma_0, \gamma_1, \cdots$ be an execution.

If there is no step $\gamma_i \mapsto \gamma_{i+1}$ in e, where there is a non-root process in V_u which is an alive abnormal root in γ_i, but not in γ_{i+1}, then the *first u-segment* of e is e itself and there is no other u-segment.

Otherwise, let $\gamma_i \mapsto \gamma_{i+1}$ be the first step of e, where there is a non-root process in V_u which is an alive abnormal root in γ_i, but not in γ_{i+1}. The *first u-segment* of e is the prefix $\gamma_0, \cdots, \gamma_{i+1}$. The *second u-segment* of e is the first u-segment of the suffix $\gamma_{i+1}, \gamma_{i+2}, \cdots$, and so forth.

By Lemma 15, we have

▶ **Observation 17.** *For every non-root process u; for every execution e, e contains at most $n_{\texttt{maxCC}} + 1$ u-segments, because there are initially at most $n_{\texttt{maxCC}}$ alive abnormal roots in V_u.*

▶ **Lemma 18.** *Let u be any non-root process. During a u-segment, if u executes the rule $\mathrm{R_{EF}}$, then u does not execute any other rule in the remaining of the u-segment.*

Proof. Let seg_u be a u-segment. Let s_1 be a step of seg_u in which u executes $\mathrm{R_{EF}}$. Let s_2 be the next step in which u executes its next rule. (If s_1 or s_2 do not exist, then the lemma trivially holds for seg_u.) Just before s_1, all branches containing u have an alive abnormal root, namely the non-root process v at depth 1 in any of these branches. (Note that we may have $v = u$.) On the other hand, just before s_2, u is the dead abnormal root of all branches it belongs to. This implies that v must have executed the rule $\mathrm{R_{EF}}$ in the meantime and thus is not an alive abnormal root anymore when the step s_2 is executed. Therefore, s_1 and s_2 belong to two distinct u-segments of the execution. ◀

▶ **Corollary 19.** *Let u be a non-root process. The sequence of rules executed by u during a u-segment belongs to the following language: $(\mathrm{R_I} + \varepsilon)(\mathrm{R_R} + \varepsilon){\mathrm{R_C}}^*(\mathrm{R_{EB}} + \varepsilon)(\mathrm{R_{EF}} + \varepsilon)$.*

We use the notion of *maximal causal chain* to further analyze the number of steps in a u-segment.

▶ **Definition 20** (Maximal Causal Chain)**.** Let u be a non-root process and seg_u be any u-segment. A *maximal causal chain of* seg_u rooted at $u_0 \in V_u$ is a maximal sequence of actions $a_1, a_2, \cdots, a_k$ executed in seg_u such that the action a_1 sets par_{u_1} to $u_0 \in V_u$ not later than any other action by u_0 in seg_u, and for all $2 \leq i \leq k$, the action a_i sets par_{u_i} to u_{i-1} after the action a_{i-1} but not later than u_{i-1}'s next action.

▶ **Observation 21.**

- *An action a_i belongs to a maximal causal chain if and only if a_i consists in a call to the macro computePath by a non-root process.*
- *Only actions of Rules $\mathrm{R_R}$ and $\mathrm{R_C}$ contain the execution of computePath.*

Let u be a non-root process and seg_u be any u-segment. Let $a_1, a_2, \cdots, a_k$ be a maximal causal chain of seg_u rooted at u_0.

- *For all $1 \leq i \leq k$, a_i consists in the execution of computePath by u_i (*i.e.*, u_i executes the rule $\mathrm{R_R}$ or $\mathrm{R_C}$) where $u_i \in V_u$.*
- *Denote by $ds_{\mathsf{seg}_u,v}$ the distance value of process v at the beginning of seg_u. For all $1 \leq i \leq k$, a_i sets d_{u_i} to $ds_{\mathsf{seg}_u,u_0} + \sum_{j=1}^{j=i} w(u_j, u_{j-1})$.*

For the next lemmas and theorems, we recall that $n_{\texttt{maxCC}} \leq n - 1$ is the maximum number of non-root processes in a connected component of G.

▶ **Lemma 22.** *Let u be a non-root process. All actions in a maximal causal chain of a u-segment are caused by different non-root processes of V_u. Moreover, an execution of computePath by some non-root process v never belongs to any maximal causal chain rooted at v.*

Proof. First note that any rule $\mathrm{R_C}$ executed by a process v makes the value of d_v decrease.

Assume now, by the contradiction, that there exists a process v such that, in some maximal causal chain $a_1, a_2, \cdots, a_k$ of a u-segment, v is used as parent in some action a_i and executes the action a_j, with $j > i$. The value of d_v is strictly larger just after the action a_j than just before the action a_i. This implies that process v must have executed the rule $\mathrm{R_R}$ in the meantime. So, a_i and a_j are executed in two different u-segments by Corollary 19 and the fact that v has status C just before the action a_i. Consequently, they do not belong to the same maximal causal chain, a contradiction.

Therefore, all actions in a maximal causal chain are caused by different processes, and a process never executes an action in a maximal causal chain it is the root of. As all actions in a maximal causal chain are executed by process in the same connected component, we are done. ◀

▶ **Definition 23** ($S_{\mathsf{seg}_u,v}$)**.** Given a non-root process u and a u-segment seg_u, we define $S_{\mathsf{seg}_u,v}$ as the set of all the distance values obtained after executing an action belonging to any maximal causal chain of seg_u rooted at process v ($v \in V_u$)).

▶ **Lemma 24.** *Given a non-root process u and a u-segment seg_u, if the size of $S_{\mathsf{seg}_u,v}$ is bounded by X for all process $v \in V_u$, then the number of computePath executions done by u in seg_u is bounded by $X(n_{maxCC} - 1)$.*

Proof. Except possibly the first, all *computePath* executions done by a u in a u-segment seg_u are done through the rule $\mathrm{R_C}$. For all these, the variable d_u is always decreasing. Therefore, all the values of d_u obtained by the *computePath* executions done by u are different. By definition of $S_{\mathsf{seg}_u,v}$ and by Lemma 22, all these values belong to the set $\bigcup_{v \in V_u \setminus \{y\}} S_{\mathsf{seg}_u,v}$, which has size at most $X(n_{\mathtt{maxCC}} - 1)$. ◀

By definition, each step contains at least one action, made by a non-root process. Let u be any non-root process. Assume that, in any u-segment seg_u, the size of $S_{\mathsf{seg}_u,v}$ is bounded by X for all process $v \in V_u$. So, the number of step of u in seg_u is bounded by $X(n_{\mathtt{maxCC}} - 1) + 3$, by Lemma 24 and Corollary 19. Moreover, recall that each execution contains at most $n_{\mathtt{maxCC}} + 1$ u-segments (Observation 17). So, u executes in at most $X{n_{\mathtt{maxCC}}}^2 + 3n_{\mathtt{maxCC}} - X + 3$ steps. Finally, as u is an arbitrary non-root process and there are $n - 1$ non-root processes, follows.

▶ **Theorem 25.** *If the size of $S_{\mathsf{seg}_u,v}$ is bounded by X for all non-root process u, for all u-segment seg_u, and for all process v in V_u, then the total number of steps during any execution, is bounded by $(X{n_{maxCC}}^2 + 3n_{maxCC} - X + 3)(n - 1)$.*

Let $\mathtt{W}_{\max} = \max_{\{u,v\} \in E} \omega(u, v)$. The size of any $S_{\mathsf{seg}_u,v}$, where u is a non-root process and $v \in V_u$, is bounded by $\mathtt{W}_{\max} n_{\mathtt{maxCC}}$, because $S_{\mathsf{seg}_u,v} \subseteq [ds_{\mathsf{seg}_u,v} + 1, ds_{\mathsf{seg}_u,v} + \mathtt{W}_{\max}(n_{cc} - 1)]$, where $n_{cc} \leq n_{\mathtt{maxCC}} + 1$ is the number of processes in V_u. Hence, we deduce the following theorem from Theorem 25 and Corollary 14.

▶ **Theorem 26.** *Algorithm* RSP *is silent self-stabilizing under the distributed unfair daemon for the set $\mathcal{LC}_{\mathsf{RSP}}$ and its stabilization time in steps is at most $[\mathtt{W}_{\max}{n_{maxCC}}^3 + (3 - \mathtt{W}_{\max})n_{maxCC} + 3](n - 1)$, i.e., $O(\mathtt{W}_{\max}{n_{maxCC}}^3 n)$.*

If all edges in G have the same weight w, then the size of $S_{\mathsf{seg}_u,v}$, where u is a non-root process and $v \in V_u$, is bounded by $n_{\mathtt{maxCC}}$. Indeed, in such a case, we have $S_{\mathsf{seg}_u,v} \subset \{ds_{\mathsf{seg}_u,v} + i.w \mid 1 \leq i \leq n_{cc} - 1\}$, where $n_{cc} \leq n_{\mathtt{maxCC}} + 1$ is the number of processes in V_u. Hence, we obtain the following corollary.

▶ **Corollary 27.** *If all edges have the same weight, then the stabilization time in steps of Algorithm* RSP *is at most* $({n_{maxCC}}^3 + 2n_{maxCC} + 3)(n-1)$*, which is less than or equal to* n^4 *for all* $n \geq 2$*.*

5 Round Complexity of Algorithm RSP

We now prove that every execution of Algorithm RSP lasts at most $3n_{\texttt{maxCC}} + D$ rounds, where $n_{\texttt{maxCC}}$ is the maximum number of non-root processes in a connected component and D is the hop-diameter of the connected component containing r, V_r.

The first lemma essentially claims that all processes that are in illegal branches progressively switch to status EB within $n_{\texttt{maxCC}}$ rounds, in order of increasing depth.

▶ **Lemma 28.** *Let* $i \in \mathbb{N}^*$*. Starting from the beginning of round* i*, there does not exist any process both in state* C *and at depth less than* i *in an illegal branch.*

Proof. We prove this lemma by induction on i. The base case ($i = 1$) is obvious, so we assume that the lemma holds for some integer $i \geq 1$. From the beginning of round i, no process can ever choose a parent which is at depth smaller than i in an illegal branch because those processes will never have status C, by induction hypothesis. Moreover, no process with status C can have its depth decreasing to i or smaller by an action of one of its ancestors at depth smaller than i, because these processes have status EB and have at least one child not having status EF. Thus, they cannot execute any rule. Therefore, no process can take state C at depth smaller or equal to i in an illegal branch.

Consider any process u with status C at depth i in an illegal branch at the beginning of the round i. $u \neq r$. Moreover, by induction hypothesis, u is an abnormal root, or the parent of u is not in state C (*i.e.*, it is in the state EB). During round i, u will execute rule $\mathrm{R_{EB}}$ or $\mathrm{R_C}$ and thus either switch to state EB or join another branch at a depth greater than i. This concludes the proof of the lemma. ◀

▶ **Corollary 29.** *After at most* n_{maxCC} *rounds, the system is in a configuration from which no process in any illegal branch has status* C *forever.*

Moreover, once such a configuration is reached, each time a process executes a rule other than $\mathrm{R_{EF}}$*, this process is outside any illegal branch forever.*

The next lemma essentially claims that, once no process in an illegal branch has status C forever, processes in illegal branches progressively switch to status EF within at most $n_{\texttt{maxCC}}$ rounds, in order of decreasing depth.

▶ **Lemma 30.** *Let* $i \in \mathbb{N}^*$*. Starting from the beginning of round* $n_{maxCC} + i$*, there does not exist any process at depth larger than* $n_{maxCC} - i + 1$ *in an illegal branch having the status* EB*.*

Proof. We prove this lemma by induction on i. The base case ($i = 1$) is obvious, so we assume that the lemma holds for some integer $i \geq 1$. By induction hypothesis, at the beginning of round $n_{\texttt{maxCC}} + i$, no process at depth larger than $n_{\texttt{maxCC}} - i + 1$ has the status EB. Therefore, processes with status EB at depth $n_{\texttt{maxCC}} - i + 1$ in an illegal branch can execute the rule $\mathrm{R_{EF}}$ at the beginning of round $n_{\texttt{maxCC}} + i$. These processes will thus all execute within round $n_{\texttt{maxCC}} + i$ (they cannot be neutralized as no children can connect to them). We conclude the proof by noticing that, from Corollary 29, once round $n_{\texttt{maxCC}}$ has terminated, any process in an illegal branch that executes either gets status EF, or will be outside any illegal branch forever. ◀

The next lemma essentially claims that, after the propagation of status EF in illegal branches, the maximum length of illegal branches progressively decreases until all illegal branches vanish.

▶ **Lemma 31.** *Let $i \in \mathbb{N}^*$. Starting from the beginning of round $2n_{maxCC} + i$, there does not exist any process at depth larger than $n_{maxCC} - i + 1$ in an illegal branch.*

Proof. We prove this lemma by induction on i. The base case ($i = 1$) is obvious, so we assume that the lemma holds for some integer $i \geq 1$. By induction hypothesis, at the beginning of round $2n_{\texttt{maxCC}} + i$, no process is at depth larger than or equal to $n_{\texttt{maxCC}} - i + 1$ in an illegal branch. All processes in an illegal branch have the status EF. So, at the beginning of round $2n_{\texttt{maxCC}} + i$, any abnormal root satisfies the predicate P_reset, they are enabled to execute either $\mathrm{R_I}$, or $\mathrm{R_R}$. So, all abnormal roots at the beginning of the round $2n_{\texttt{maxCC}} + i$ are no more in an illegal branch at the end of this round: the maximal depth of the illegal branches has decreased, since by Corollary 29, no process can join an illegal tree during the round $2n_{\texttt{maxCC}} + i$. ◀

▶ **Corollary 32.** *After at most round $3n_{maxCC}$, there are no illegal branches forever.*

Note that in any connected component that does not contain the root r, there is no legal branch. Then, since the only way for a process to be in no branch is to have status I, we obtain the following corollary.

▶ **Corollary 33.** *For any connected component H other than V_r, after at most $3n_{maxCC}$ rounds, every process of H is in a legitimate state forever.*

In the connected component V_r, Algorithm RSP may need additional rounds to propagate the correct distances to r. In the next lemma, we use the notion of hop-distance to r defined below.

▶ **Definition 34** (Hop-Distance and Hop-Diameter)**.** A process u is said to be at *hop-distance* k from v if the minimum number of edges in a shortest path from u to v is k.

The *hop-diameter* of a graph G (resp. of a connected component H of the graph G) is the maximum hop-distance between any two nodes of G (resp. of H).

▶ **Lemma 35.** *Let $i \in \mathbb{N}$. In every execution of Algorithm* RSP*, starting from the beginning of round $3n_{maxCC} + i$, every process at hop-distance at most i from r is in a legitimate state.*

Proof. We prove this lemma by induction on i. First, by definition, the root r is always in a legitimate state, so the base case ($i = 0$) trivially holds. Then, after at most $3n_{\texttt{maxCC}}$ rounds, every process either belongs to a legal branch or has status I (by Corollary 32), thus any non-isolated process $v \in V_r$ always stores a distance d such that $d \geq d(v, r)$, its actual weighted distance to r. By induction hypothesis, every process at hop-distance at most i from r has converged to a legitimate state within at most $3n_{\texttt{maxCC}} + i$ rounds. Therefore, at the beginning of round $3n_{\texttt{maxCC}} + i + 1$, every process v at hop-distance $i + 1$ from r which is not in a legitimate state is enabled for executing rule $\mathrm{R_C}$. Thus, at the end of round $3n_{\texttt{maxCC}} + i + 1$, every process at hop-distance at most $i + 1$ from r is in a legitimate state (such processes cannot be neutralized during this round). Also, these processes will never change their state since there are no processes that can make them closer to r. ◀

Summarizing all the results of this section, we obtain the following theorem.

▶ **Theorem 36.** *Every execution of Algorithm* RSP *lasts at most* $3n_{maxCC} + D$ *rounds, where* n_{maxCC} *is the maximum number of non-root processes in a connected component and* D *is the hop-diameter of the connected component containing* r*.*

Recall that under a *weakly fair* daemon, every continuously enabled process is eventually activated by the daemon. By definition, every round is finite, yet maybe unbounded, in terms of steps under such an assumption. Hence, if every execution contains a finite number of rounds, then every execution is finite under the weakly fair daemon assumption.

Notice then that all proofs made in this section still hold if we assume that edge weights are strictly positive real numbers. Hence

▶ **Observation 37.** *If edge weights are strictly positive real numbers, then Algorithm* RSP *is silent self-stabilizing under the distributed weakly fair daemon for the set* $\mathcal{LC}_{\mathsf{RSP}}$ *and its stabilization time in rounds is still at most* $3n_{maxCC} + D$*, where* n_{maxCC} *is the maximum number of non-root processes in a connected component and* D *is the hop-diameter of the connected component containing* r*.*

References

1. Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Information and Computation, special issue of SSS 2014*, 2016. To appear.
2. A. Arora, M. G. Gouda, and T. Herman. Composite routing protocols. In *the 2nd IEEE Symposium on Parallel and Distributed Processing (SPDP'90)*, pages 70–78, 1990.
3. Richard Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.
4. Lélia Blin, Alain Cournier, and Vincent Villain. An improved snap-stabilizing PIF algorithm. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003*, volume 2704 of *Lecture Notes in Computer Science*, pages 199–214, San Francisco, CA, USA, June 24-25 2003. Springer.
5. Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. *J. Parallel Distrib. Comput.*, 81-82:11–23, 2015. doi:10.1016/j.jpdc.2015.02.001.
6. Srinivasan Chandrasekar and Pradip K Srimani. A self-stabilizing distributed algorithm for all-pairs shortest path problem. *Parallel Algorithms and Applications*, 4(1-2):125–137, 1994.
7. Ernest J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, 1982.
8. N. S. Chen, H. P. Yu, and S. T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.
9. J. A. Cobb and M. G. Gouda. Stabilization of general loop-free routing. *Journal of Parallel and Distributed Computing*, 62(5):922–944, 2002.
10. Alain Cournier, Stéphane Devismes, and Vincent Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), 2009.
11. Alain Cournier, Stephane Rovedakis, and Vincent Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. In *the 15th International Conference on Principles of Distributed Systems (OPODIS'11), Springer LNCS 7109*, pages 159–174, 2011.
12. Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An o(n)-time self-stabilizing leader election algorithm. *jpdc*, 71(11):1532–1544, 2011.

13 Ajoy Kumar Datta, Stéphane Devismes, and Lawrence L. Larmore. Brief announcement: Self-stabilizing silent disjunction in an anonymous network. In *the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'12), Springer LNCS 7596*, pages 46–48, 2012.

14 Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011.

15 Stéphane Devismes and Colette Johnen. Silent self-stabilizing {BFS} tree algorithms revisited. *Journal of Parallel and Distributed Computing*, 97:11–23, 2016. doi:10.1016/j.jpdc.2016.06.003.

16 Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

17 S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

18 Shlomi Dolev. *Self-stabilization.* MIT Press, March 2000.

19 Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

20 Lester R. Ford Jr. Network flow theory, August 1956. Paper P-923, RAND Corporation, Santa Monica, California, USA.

21 Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. In *the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'14), Springer LNCS 8736*, pages 120–134, 2014.

22 Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks – extended version. Technical report, LaBRI, CNRS UMR 5800, 2016. URL: https://hal.archives-ouvertes.fr/hal-01352245.

23 Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

24 Tetz C. Huang. A self-stabilizing algorithm for the shortest path problem assuming the distributed demon. *Computers & Mathematics with Applications*, 50(5–6):671–681, 2005.

25 Tetz C. Huang and Ji-Cherng Lin. A self-stabilizing algorithm for the shortest path problem in a distributed system. *Computers & Mathematics with Applications*, 43(1):103–109, 2002.

26 C. Johnen and S. Tixeuil. Route preserving stabilization. In *the 6th International Symposium on Self-stabilizing System (SSS'03), Springer LNCS 2704*, pages 184–198, 2003.

27 Alberto Leon-Garcia and Indra Widjaja. *Communication Networks.* McGraw-Hill, Inc., New York, NY, USA, 2 edition, 2004.

28 Adrian Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, 29(1):23–34, 1983.

29 M. Sloman and J. Kramer. *Distributed systems and computer networks.* Prentice Hall, 1987.

30 G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, Cambridge, UK, Second edition 2001.

Polynomial Self-Stabilizing Maximum Matching Algorithm with Approximation Ratio 2/3

Johanne Cohen[1], Khaled Maâmra[2], George Manoussakis[3], and Laurence Pilard[4]

1 **LRI-CNRS, Université Paris-Sud, Université Paris Saclay, Paris, France**
johanne.cohen@lri.fr
2 **LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, Paris, France**
khaled.maamra@uvsq.fr
3 **LRI-CNRS, Université Paris-Sud, Université Paris Saclay, Paris, France**
george.manoussakis@lri.fr
4 **LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, Paris, France**
laurence.pilard@uvsq.fr

Abstract

We present the first polynomial self-stabilizing algorithm for finding a $\frac{2}{3}$-approximation of a maximum matching in a general graph. The previous best known algorithm has been presented by Manne *et al.* [16] and has a sub-exponential time complexity under the distributed adversarial daemon [3]. Our new algorithm is an adaptation of the Manne *et al.* algorithm and works under the same daemon, but with a time complexity in $O(n^3)$ moves. Moreover, our algorithm only needs one more boolean variable than the previous one, thus as in the Manne *et al.* algorithm, it only requires a constant amount of memory space (three identifiers and *two* booleans per node).

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Self-Stabilization, Distributed Algorithm, Fault Tolerance, Matching

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.11

1 Introduction

Matching problems have received a lot of attention in different areas. Dynamic load balancing and job scheduling in parallel and distributed networks can be solved by algorithms using a matching set of communication links [2, 7]. Moreover, the matching problem has been recently studied in the algorithmic game theory. Indeed, the seminal problem relative to matching introduced by Knuth is the stable marriage problem [13]. This problem can be modeled as a game used in social networks [10] and in wireless networks [18].

In graph theory, a *matching* M in a graph G is a subset of the edges of G without common nodes. A matching is *maximal* if no proper superset of M is also a matching whereas a *maximum* matching is a maximal matching with the highest cardinality among all possible maximal matchings. Some (almost) linear time approximation algorithm for the maximum weighted matching problem have been well studied [6, 17], nevertheless these algorithms are not distributed. They are based on a simple greedy strategy using *augmenting path*. An *augmenting path* is a path, starting and ending in an unmatched node, and where every other edge is either unmatched or matched; *i.e.* for each consecutive pair of edges, exactly one of them must belong to the matching. Let us consider the example in Figure 2d, page 11. In

this figure, u and v are matched nodes and x, y are unmatched nodes. The path (x, u, v, y) is an augmenting path of length 3 (written 3-*augmenting path*). It is well known [11] that given a graph $G = (V, E)$ and a matching $M \subseteq E$, if there is no augmenting path of length $2k - 1$ or less, then M is a $\frac{k}{k+1}-$approximation of the maximum matching. See [6] for the weighted version of this theorem. The greedy strategy in [6, 17] consists in finding all augmenting paths of length ℓ or less and by switching matched and unmatched edges of these paths in order to improve the maximum matching approximation.

In this paper, we present a self-stabilizing algorithm for finding a maximum matching with approximation ratio 2/3 that uses the greedy strategy presented above. Our algorithm stabilizes after $O(n^3)$ moves under the adversarial distributed daemon.

In unweighted or weighted general graphs, self-stabilizing algorithms for computing maximal matching have been designed in various models (anonymous network [1] or not [19], see [8] for a survey). For an unweighted graph, Hsu and Huang [12] gave the first self-stabilizing algorithm and proved a bound of $O(n^3)$ on the number of moves under a sequential adversarial daemon. Hedetniemi *et al.* [9] completed the complexity analysis proving a $O(m)$ move complexity. Manne *et al.* [15] gave a self-stabilizing algorithm that converges in $O(m)$ moves under a distributed adversarial daemon. Note that it is well known that a maximal matching is only an $\frac{1}{2}$-approximation of a maximum matching.

Manne *et al.* [16] and Asada and Inoue [1] presented some self-stabilizing algorithms for finding a $\frac{2}{3}$-approximation of a maximum matching. Manne *et al.* gave an exponential upper bound on the stabilization time ($O(2^n)$ moves under a distributed adversarial daemon) of their algorithm. However, they didn't show that this upper bound is tight. We proved [3] that this lower bound is sub-exponential by exhibiting an execution of $\Omega(2^{\sqrt{n}})$ moves before stabilization. Asada and Inoue [1] gave a polynomial algorithm but under the adversarial sequential daemon.

In a weighted graph, Manne and Mjelde [14] presented the first self-stabilizing algorithm for computing a weighted matching of a graph with an $\frac{1}{2}$-approximation of the optimal solution. They established that their algorithm stabilizes after at most an exponential number of moves under any adversarial daemon (*i.e.*, sequential or distributed). Turau and Hauck [19] gave a modified version of the previous algorithm that stabilizes after $O(nm)$ moves under any adversarial daemon.

The following figure compares features of the aforementioned algorithms and our result. The previous best known algorithm working under the same daemon has a sub-exponential complexity while our algorithm has a cubic complexity.

Problem	$\frac{1}{2}$-approximation (maximal matching)		$\frac{2}{3}$-approximation		
Graph	Identified		Anonymous without cycle with multiple of 3 length	Identified	
Daemon	Adversarial Sequential	Adversarial Distributed	Adversarial Sequential	Adversarial Distributed	
Reference	[12, 9]	[15]	[1]	[16]	**This paper**
Complexity	$O(m)$ moves	$O(m)$ moves	$O(m)$ moves	$\Omega(2^{\sqrt{n}})$ moves	$\mathbf{O(n^3)}$ **moves**

In the rest of the document, we present the model (Section 2), the algorithm (Section 3) and then the correction proof (Section 4) followed by the convergence proof (Section 5).

2 Model

The system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is represented by an undirected graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. Each process corresponds to a node in V and two processes u and v are adjacent if and only if $(u, v) \in E$. The set of neighbors of a process u is denoted by $N(u)$ and is the set of all processes adjacent to u, and Δ is the maximum degree of G. We assume all nodes in the system have a unique identifier.

For the communication, we consider the *shared memory model*. In this model, each process maintains a set of *local variables* that makes up the *local state* of the process. A process can read its local variables and the local variables of its neighbors, but it can write only in its own local variables. A *configuration* C is the local states of all processes in the system. Each process executes the same algorithm that consists of a set of *rules*. Each rule is of the form of <*name*> :: **if** <*guard*> **then** <*command*>. The *name* is the name of the rule. The *guard* is a predicate over the variables of both the process and its neighbors. The *command* is a sequence of actions assigning new values to the local variables of the process.

A rule is *activable* in a configuration C if its guard in C is true. A process is *eligible* for the rule $\mathcal{R}$ in a configuration C if its rule $\mathcal{R}$ is activable in C and we say the process is *activable* in C. An *execution* is an alternate sequence of configurations and actions $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, \ldots$, such that $\forall i \in \mathbb{N}^*$, C_{i+1} is obtained by executing the command of at least one rule that is activable in C_i (a process that executes such a rule makes a *move*). More precisely, A_i is the non empty set of activable rules in C_i that has been executed to reach C_{i+1} and such that each process has at most one of its rules in A_i. We use the notation $C_i \mapsto C_{i+1}$ to denote this transition in $\mathcal{E}$. Finally, let $\mathcal{E}' = C'_0, A'_0, \cdots, C'_k$ be a finite execution. We say $\mathcal{E}'$ is a *sub-execution* of $\mathcal{E}$ if and only if $\exists t \geq 0$ such that $\forall j \in [0, \cdots, k]$:$(C'_j = C_{j+t} \wedge A'_j = A_{j+t})$.

An *atomic operation* is such that no change can take place during its run, we usually assume that an atomic operation is instantaneous. In the shared memory model, a process u can read the local state of all its neighbors and update its whole local state in one atomic step. Then, we assume here that a rule is an atomic operation. An execution is *maximal* if it is infinite, or it is finite and no process is activable in the last configuration. All algorithm executions considered here are assumed to be maximal.

A *daemon* is a predicate on the executions. We consider only the most powerful one: the *adversarial distributed daemon* that allows all executions described in the previous paragraph. Observe that we do not make any fairness assumption on the executions.

An algorithm is *self-stabilizing* for a given specification, if there exists a sub-set $\mathcal{L}$ of the set of all configurations such that: every execution starting from a configuration of $\mathcal{L}$ verifies the specification (*correctness*) and starting from any configuration, every execution eventually reaches a configuration of $\mathcal{L}$ (*convergence*). $\mathcal{L}$ is called the set of *legitimate configurations*. A configuration is *stable* if no process is activable in the configuration. The algorithm presented here, is *silent*, meaning that once the algorithm has stabilized, no process is activable. In other words, all executions of a silent algorithm are finite and end in a stable configuration. Note the difference with a non silent self-stabilizing algorithm that has at least one infinite execution with a suffix only containing legitimate configurations, but not stable ones.

3 Algorithm

The algorithm presented in this paper is called MaxMatch, and is based on the algorithm presented by Manne *et al.* [16]. As in the Manne *et al.* algorithm, MaxMatch assumes there

exists an underlying maximal matching algorithm, which has reached a stable configuration. Based on this stable maximal matching, MaxMatch builds a $\frac{2}{3}$-approximation of the maximum matching by detecting and then deleting all 3-augmenting paths. Once a 3-augmenting path is detected, nodes rearrange the matching accordingly, *i.e.*, transform this path with one matched edge into a path with two matched edges. This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. The algorithm stabilizes when there is no augmenting path of length three left. By the result of Hopcroft *et al.* [11], we obtain a $\frac{2}{3}$-approximation of the maximum matching.

This underlying stabilized maximal matching can be built, for instance, with the self-stabilizing maximal matching algorithm from [15] that stabilizes in $O(m)$ moves under the adversarial distributed daemon (so the same daemon than the one used in this paper). Observe that this algorithm is silent, meaning that the maximal matching remains constant once the algorithm has stabilized. Then, using a classical composition of these two algorithms [5], we obtain a total time complexity in $O(n^2 \times n^3) = O(n^5)$ moves under the adversarial distributed daemon.

In the rest of the paper, $\mathcal{M}$ is the underlying maximal matching, and $\mathcal{M}^+$ is the set of edges built by our algorithm MaxMatch (see Definition 3.1). For a set of nodes A, we define $single(A)$ and $matched(A)$ as the set of unmatched and matched nodes in A, accordingly to the underlying maximal matching $\mathcal{M}$. Moreover, $\mathcal{M}$ is encoded with the variable m_u. If $(u, v) \in \mathcal{M}$ then u and v are *matched nodes* and we have: $m_u = v \wedge m_v = u$. If u is not incident to any edge in $\mathcal{M}$, then u is a *single node* and $m_u = null$. Since we assume the underlying maximal matching is stable, a node membership in $matched(V)$ or $single(V)$ will not change, and each node u can use the value of m_u to determine which set it belongs to.

Variables description: In order to facilitate the rematching, each node $u \in V$ maintains three pointers and two boolean variables. The pointer p_u refers to a neighbor of u that u is trying to (re)match with. If $p_u = null$ then the matching of u has not changed from the maximal matching. Thus, the matching $\mathcal{M}^+$ built by our algorithm is defined as follows:

▶ **Definition 3.1.** The set of edges built by algorithm MaxMatch is $\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$

For a matched node u, pointers α_u and β_u refer to two nodes in $single(N(u))$ that are *candidates* for a possible rematching with u. Also, s_u is a boolean variable that indicates if the node u has performed a successful rematching with its single node candidate. Finally, end_u is a boolean variable that indicates if both u and m_u have performed a successful rematching or not. For a single node x, only one pointer p_x and one boolean variable end_x are needed. p_x has the same purpose as the p-variable of a matched node. The *end*-variable of a single node allows the matched nodes to know whether it is *available* or not. A single node is *available* if it is possible for this node to eventually rematch with one of its neighboring married node, *i.e.*, end_x =*False*.

In our algorithm, $Unique(A)$ returns the number of unique elements in the multi-set A, and $Lowest(A)$ returns the node in A with the lowest identifier. If $A = \emptyset$, then $Lowest(A)$ returns $null$. Moreover, rules have priorities. In the algorithm, we present rules from the highest priority (at the top) to the lowest one (at the bottom).

Algorithm description: Every pair of matched nodes u, v (v=m_u) tries to find single neighbors they can rematch with. Moreover u and v need to have a sufficient number of available single neighbors to detect a 3-augmenting path: each node should have at least

——— **Rules for each node u in single($\mathbf{V}$)**

ResetEnd ::
if $p_u = null \wedge end_u = True$
then $end_u := False$

UpdateP ::
if $(p_u = null \wedge \{w \in matched(N(u)) \mid p_w = u\} \neq \emptyset) \vee (p_u \notin (matched(N(u)) \cup \{null\})) \vee$
$(p_u \neq null \wedge p_{p_u} \neq u)$
then $p_u := Lowest\{w \in N(u) \mid p_w = u\}$
$end_u := False$

UpdateEnd ::
if $(p_u \in matched(N(u)) \wedge (p_{p_u} = u) \wedge (end_u \neq end_{p_u})$
then $end_u := end_{p_u}$

——— **Predicates and functions**

BestRematch$(u) \equiv$
$a = Lowest\{x \in single(N(u)) \wedge (p_x = u \vee end_x = False)\}$
$b = Lowest\{x \in single(N(u)) \setminus \{a\} \wedge (p_x = u \vee end_x = False)\}$
return (a, b)

AskFirst$(u) \equiv$
if $\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\})$
then if $(\alpha_u < \alpha_{m_u}) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_u = null) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_{m_u} \neq null \wedge u < m_u)$
then return α_u
else return $null$

AskSecond$(u) \equiv$
if $AskFirst(m_u) \neq null$
then return $Lowest(\{\alpha_u, \beta_u\} \setminus \{\alpha_{m_u}\})$
else return $null$

——— **Rules for each node u in matched($\mathbf{V}$)**

Update ::
if $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null)$
$\vee p_u \notin (single(N(u)) \cup \{null\}) \vee$
$((\alpha_u, \beta_u) \neq BestRematch(u) \wedge (p_u = null \ \vee \ (p_{p_u} \neq u \ \wedge \ end_{p_u} = True)))$
then $(\alpha_u, \beta_u) := BestRematch(u)$
$(p_u, s_u, end_u) := (null, False, False)$

MatchFirst ::
if $(AskFirst(u) \neq null) \wedge$
$[\ p_u \neq AskFirst(u) \vee$
$s_u \neq (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_{m_u} \in \{AskSecond(m_u), null\}) \vee$
$end_u \neq (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})\ ]$
then $end_u := (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})$
$s_u := (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge (p_{m_u} \in \{AskSecond(m_u), null\})$
$p_u := AskFirst(u)$

MatchSecond ::
if $(AskSecond(u) \neq null) \wedge (s_{m_u} = True) \wedge$
$[\ p_u \neq AskSecond(u) \vee end_u \neq (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u))$
$\vee s_u \neq end_u\ ]$
then $end_u := (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u))$
$s_u := end_u$
$p_u := AskSecond(u)$

ResetMatch ::
if $[(AskFirst(u) = AskSecond(u) = null) \wedge ((p_u, s_u, end_u) \neq (null, False, False))] \vee$
$[AskSecond(u) \neq null \wedge p_u \neq null \wedge s_{m_u} = False]$
then $(p_u, s_u, end_u) := (null, False, False)$

one candidate and the sum of the number of candidates for u and v should be at least 2. The *BestRematch* predicate is used to compute candidates in variables α and β, and the condition $(\alpha_u \neq null \wedge \alpha_v \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_v, \beta_v\}))$, from predicate *AskFirst*, is used to ensure the number of candidates is sufficiently high.

Observe that we only have three distinct possible values for the quadruplet $(AskFirst(u), AskSecond(u), AskFirst(v), AskSecond(v))$ for any couple $(u, v) \in \mathcal{M}$ and whatever the α and β values are. These are: $(null, null, null, null)$ or $(x, null, null, y)$ or $(null, x, y, null)$, with $x \neq y$. The first case means that there is no 3-augmenting path that contains the couple (u, v). The two other cases mean that (x, u, v, y) is a 3-augmenting path. If $x < y$, we are in the second case, otherwise we are in the third case. Node u is said to be *First* if $AskFirst(u) \neq null$. In the same way, u is *Second* if $AskSecond(u) \neq null$.

In the following, we consider the second case, since the third case is symmetric of the second case with v is *First*. So we assume u is *First*.

A 3-augmenting path is exploited in three phases. The first phase determines the first edge of the 3-augmenting path. This phase is complete when $s_u = True$ and $p_{p_u} = u$ (u is *First*). In the second phase, the third edge of the augmenting path is marked. This phase is complete when node v, that is *Second*, sets its *end* value to *True*. Finally, in the third phase, the *end* value of v is propagated in the whole augmenting path. When this propagation is done, the phase is over, the augmenting path is said to be *fully exploited* and all neighbors of single nodes of this path know that these two single nodes are not available anymore.

The scenario for an augmenting path exploitation when everything goes well is given in the following. Node u starts trying to rematch with x performing a *MatchFirst* move and $p_u := x$. If x accepts the proposition, performing an *UpdateP* move and $p_x := u$, then u will inform v of this first phase success, once again by performing a *MatchFirst* move and $s_u :=$*True*. Observe that at this point, x cannot change its p-value since $p_{p_x} = x$. Finally, node v tries to rematch with y, performing a *MatchSecond* move and $p_v := y$. If y accepts the proposition, performing an *UpdateP* move and $p_y := v$, then v will inform u of this final success, by performing a *MatchSecond* move and $end_v :=$*True*. This complete the second phase. From then, all nodes in this 3-augmenting path will set there *end*-variable to *True*: u by performing a last *MatchFirst* move, and x and y by performing an *UpdateEnd* move. From this point, non of nodes x, u, v, or y will ever be eligible for any move again. Moreover, once single nodes have their *end*-variables set to *True*, they are not available anymore for any other matched nodes.

Rules description: The *Update* rule allows a matched node to update its α and β variables. Then, predicates *AskFirst* and *AskSecond* are used to define the role the node will have in the 3-augmenting path exploitation. If the node is *First* (resp. *Second*), then it will execute *MatchFirst* (resp. *MatchSecond*) several times for this 3-augmenting path exploitation.

The *MatchFirst* rule is used by the node when it is *First*. Let u be this node. The first time this rule is performed, u seduces its candidate setting (end_u, s_u, p_u) to $(False, False, AskFirst(u))$. Then this rule is performed a second time after the u's candidate has accepted the u's proposition, *i.e.*, when $AskFirst(u)$ has set its p-variable to u. So the second *MatchFirst* execution sets (end_u, s_u, p_u) to $(False, True, AskFirst(u))$. Now, variable s_u is equal to $True$, allowing node m_u that is *Second* to seduce its own candidate. Finally, the rule *MatchFirst* is performed a third time when m_u completed is own rematch, *i.e.*, when $end_u = True$. When there is no bad information due to some bad initializations, $end_{m_u} = True$ means that $p_{m_u} = AskSecond(m_u) \wedge p_{p_{m_u}} = m_u$. So this third *MatchFirst* execution sets (end_u, s_u, p_u) to $(True, True, AskFirst(u))$, meaning that the 3-augmenting path has been fully exploited.

In the *MatchFirst* rule, observe that we make the s_u affectation before the p_u affectation, because the s_u value must be computed accordingly to the value of p_u before activating u. Indeed, when u executes *MatchFirst* for the first time, it allows to set p_u from $\perp$ to $AskFirst(u)$ while s_u remains $False$. Then when u executes *MatchFirst* for the second time, s_u is set from $False$ to $True$ while p_u remains equal to $AskFirst(u)$. For the same argument, we make the end_u affectation before the s_u affectation. Thus, the «normal» values sequence for (p_u, s_u, end_u) is: $((\perp, False, False), (AskFirst(u), False, False), (AskFirst(u), True, False), (AskFirst(u), True, True))$.

The *MatchSecond* rule is used by the node when it is *Second*. This rule is performed only twice in one 3-augmenting path exploitation. For the first execution, u has to wait for m_u to set its s_{m_u} to *True*. Then u can perform *MatchSecond* and update its p-variable to $AskSecond(u)$. When the u's candidate has accepted to this proposition, u can perform *MatchSecond* for the second time, setting s_u and end_u to *True*. As in the *MatchFirst* rule, we set *end* and s affectation before the p affectation.

The *ResetMatch* rule is performed to reset bad initialization and its consequences.

Let us now consider rules for single nodes. The *ResetEnd* rule is used to reset bad initializations. In the *UpdateP* rule, the node updates its p-value according to the propositions done by neighboring matched nodes. If there is no proposition, the node sets its p-value to $\perp$. Otherwise, p is set to the minimum identifier among all proposals. Afterward, the p-value can only change when the proposition is canceled. When a single node u has accepted a proposition, its end value should be equal to the end value of p_u. The *UpdateEnd* rule is used for this purpose.

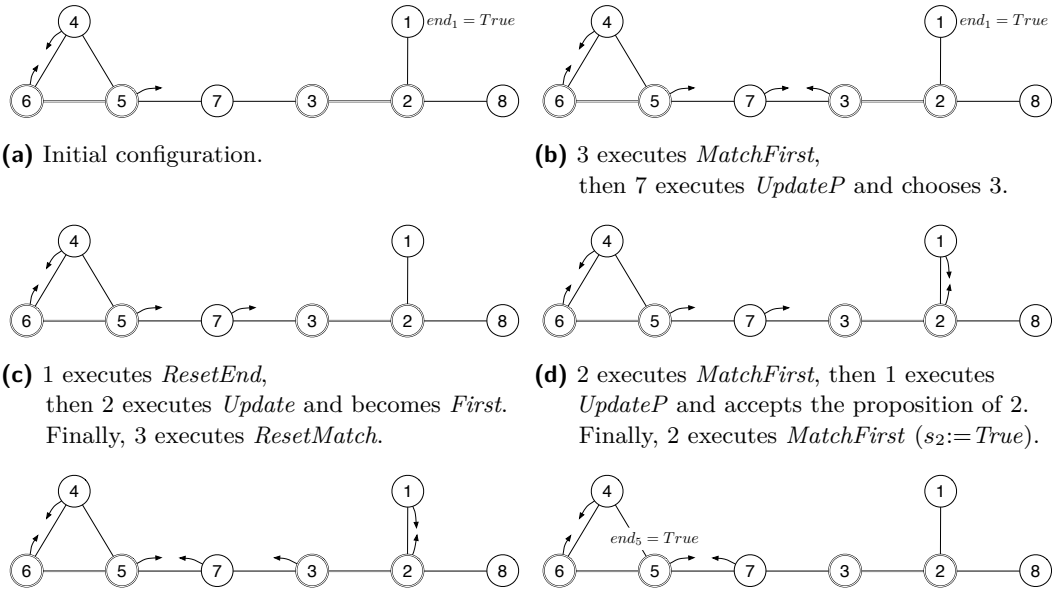
Graphical convention: We will follow the above conventions in all the figures: matched nodes are represented with double circles and single nodes with simple circles. Moreover, all edges that belong to the maximal matching $\mathcal{M}$ are represented with a double line, whereas the other edges are represented with a simple line. Black arrows show the content of the local variable p. If the p-value is *null*, we draw a 'T'. A prohibited value is first drawn in grey, then scratched out in black. If there is no knowledge on the p-value, nothing is drawn. For instance, in Figure 2e, page 11, x is a single node, u and v are matched nodes and $(u, v) \in \mathcal{M}$, $p_u = x$, and $p_x \neq u$. In Figure 2d page 11, $p_u = \perp$.

3.1 Execution example

Now, we give a possible execution of Algorithm MaxMatch under the adversarial distributed daemon. Figure 1a shows the initial state of the execution. Node identifiers are indicated inside the circles. The underlying maximal matching contains two edges $(5, 6)$ and $(2, 3)$. Then nodes 2, 3, 5 and 6 are *matched* nodes and nodes 1, 4, 7 and 8 are *single* nodes. At the beginning, there are three 3-augmenting paths: $(4, 6, 5, 7), (7, 3, 2, 1)$ and $(7, 3, 2, 8)$.

The initial configuration (Figure 1a): In the initial configuration, we assume that all α-*values* and β-*values* are defined as follows: $(\alpha_2, \beta_2) = (8, null)$, $(\alpha_3, \beta_3) = (7, null)$, $(\alpha_5, \beta_5) = (4, 7)$, and $(\alpha_6, \beta_6) = (4, null)$. We also assume all s-*values* are well defined ($s_6 =$*True* while all other s-values are *False*) whereas all *end*-values are *False* but end_1 that is *True*. At this moment, node 2 considers that since $end_1 =$*True*, node 1 already belongs to a fully exploited 3-augmenting path: $BestRematch(2) = (8, null)$.

Nodes 6 and 5 have already started to exploit their augmenting path: $p_6 = 4$, $p_4 = 6$ and $p_5 = 7$. Node 6 is *First* because $\beta_6 = null$ and since $s_6 =$*True*, node 5 knows that it can start its exploitation too, performing a *MatchSecond*: $p_5 = 7$. At this step, node 5 waits for



(a) Initial configuration.

(b) 3 executes *MatchFirst*, then 7 executes *UpdateP* and chooses 3.

(c) 1 executes *ResetEnd*, then 2 executes *Update* and becomes *First*. Finally, 3 executes *ResetMatch*.

(d) 2 executes *MatchFirst*, then 1 executes *UpdateP* and accepts the proposition of 2. Finally, 2 executes *MatchFirst* (s_2:=*True*).

(e) 3 and 7 execute in parallel *MatchSecond* and *UpdateP* respectively.

(f) 5 executes *MatchSecond*, then the *True* value of end_5 is propagated in the path $(4, 6, 5, 7)$. Finally the path $(7, 3, 2, 1)$ is destroyed.

Figure 1 An execution of Algorithm MaxMatch (Only the *True* value of the *end*-variables are given).

an answer of node 7. There are only two kinds of answer: node 7 accepts to take part of this path exploitation setting $p_7 = 5$ with an *UpdateP* rule, or it refuses setting $end_7 =$*True* while $p_7 \neq 5$ with an *UpdateEnd* rule. But this last choice can only be done if 7 belongs to another fully exploited augmenting path. So at this point, node 7 cannot refuse.

The other 3-augmenting path is $(7, 3, 2, 8)$. Node 2 considers that node 1 is not available because $end_1 =$*True*. Since $2 \leq Unique(\{\alpha_2, \beta_2, \alpha_3, \beta_3\}) \leq 4$, nodes 2 and 3 detect a 3-augmenting path and start to exploit it. Since node 3 is *First* ($AskFirst(3) = 7$ and $AskFirst(2) = null$), node 3 may execute a *MatchFirst* move. Let us assume it does.

The 3-augmenting path exploitation starts (Figure 1b): Node 3 executes here a *MatchFirst* move and points to node 7. Since both nodes 3 and 5 are pointing to node 7, node 7 can choose the node to match with from these two nodes. Note that at this point, node 7 is the only activable node among all nodes except node 1. Node 7 makes this choice by executing an *UpdateP* move: since $Lowest\{u \in N(7) \mid p_u = 7\} = 3$, node 7 points to node 3.

Difference with Manne *et al.* algorithm: In our algorithm, even after 7 has chosen 3, node 5 still waits for an acceptation of node 7, and will do so while end_7 remains *False*. However, at this point, in Manne *et al.* algorithm, node 5 can destroy the augmenting-path construction. This is the main difference that allows our algorithm to prevent from exponential executions.

So, at this point there is a binary choice for node 5: destroy or not its augmenting-path construction. In Manne *et al.* algorithm, the choice is to destroy, thus the destruction of a partially exploited augmenting-path can be done while no fully exploited augmenting path has been built. Moreover, for one fully exploited augmented path, we can exhibit some executions where we destroy a sub-exponential number of exploited augmented-path [3]. In

our algorithm, we do the other choice which is: do not destroy while there is still hope to exploit the augmenting path. So, if node 5 breaks a partially exploited augmenting path, then node 7 belongs to a fully exploited augmenting-path. Thus the destruction of 5 implies one 3-augmenting path has been fully exploited and thus the matching size has been increased by 1.

This difference is implemented in the algorithm through the *BestRematch* predicate. The condition $p_x = null$ in Manne *et al.* algorithm has been replaced by the condition $end_x = False$ in our algorithm. Then, in our algorithm, $BestRematch(5)$ remains constant when 7 chooses node 3, while it does not in Manne *et al.* algorithm, making 5 eligible for *Update*.

Go back to the execution, node 1 wakes up (Figure 1c): Let us focus on node 1. Its *end*-value is not well defined since $end_1 = True$ while node 1 does not belong to a fully exploited augmenting path. Thus, node 1 is eligible for *ResetEnd* rule. After this activation, $end_1 = False$. This implies that $BestRematch(2) = (1, 8)$ and thus $(\alpha_2, \beta_2) = (8, null) \neq BestRematch(2)$. So, node 2 is eligible for *Update* rule. Let us assume it makes this move. Thus, after this activation, node 2 is *First*. This implies that node 3 is *Second*, and it is eligible for *ResetMatch* because $AskSecond(3) \neq null \wedge p_3 \neq null \wedge s_2 = False$. So, it does it and sets $p_3 = null$.

A second 3-augmenting path exploitation starts (Figure 1d): Let us consider node 2. It is *First* and it can execute a *MatchFirst* rule. After this activation, it sets $p_2 = 1$ and $s_2 = end_2 = False$. Now, node 1 accepts the node 2 proposition by applying *UpdateP*. After this activation, node 1 points to node 2 ($p_1 = 2$). Now, node 2 is eligible for executing a *MatchFirst* rule. It sets $p_2 = 1$ and $s_2 = True$. This implies that node 3 becomes eligible for *MatchSecond*.

A matched node proposition in parallel with a single node abandonment (Figure 1e): In the configuration shown in Figure 1d, only nodes 3 and 7 are activable, node 3 can propose to node 7 with a *MatchSecond* and node 7 can accept the node 5's proposition with an *UpdateP*. Assume 3 and 7 are activated at the same time. Figure 1e shows the configuration obtained after theses moves: $p_3 = 7$, $p_7 = 5$. Note that after these activations, we have $s_3 = False$ since, *before* these activations, the *p*-values of nodes 3 and 7 are not as follow: $p_3 = 7$ and $p_7 = 3$. This kind of transitions, where a matched node proposition is performed in parallel with a single node abandonment, is the reason why we make the *s*-affectation, then the *p*-affectation in the *MatchFirst* rule. This trick allows to obtain after a *MatchFirst* rule: $s_u = True$ implies $p_{p_u} = u$. Finally, observe at this step that node 3 still waits for an answer of node 7.

The path (4,6,5,7) becomes fully exploited (Figure 1f): Since $end_5 \neq (p_5 = AskSecond(5) \wedge p_7 = 5 \wedge p_6 = AskFirst(6))$, node 5 is eligible for a *MatchSecond* rule to set end_5 to *True* and then to make the other nodes aware that the path is fully exploited. Assume node 5 executes a *MatchSecond* move. This will cause node 7 (resp. 6) to execute an *UpdateEnd* move (resp. a *MatchFirst* move) and sets $end_7 = True$ (resp. $end_6 = True$). Now, it is the turn to node 4 to execute an *UpdateEnd* move. As the *end*-value of nodes 4, 5, 6, and 7 are equal to *True*, the 3-augmenting path is fully exploited.

Recall that node 3 was waited for an answer from node 7. Now, $end_7 = True \wedge p_7 \neq 3$. Thus node 7 is not available for node 3 anymore and so node 3 executes the *Update* rule:

$(\alpha_3, \beta_3) = (null, null)$. This will cause node 2 to execute a *ResetMatch* move ($p_2 = null$) and then node 1 to execute an *UpdateP* move ($p_1 = null$). The system has reached a stable configuration (see Fig. 1e). Thus, the size of the matching is increasing by one and there is no 3-augmenting path left.

Now, we present the proof of our algorithm.

4 Correctness Proof

In all the proofs, if a lemma, a theorem, or a corollary is labeled with a capital letter, then the associated proof is in [4].

A natural way to prove the correction of MaxMatch algorithm could have been to follow the approach below. We consider a stable configuration C in MaxMatch and we prove C is also stable in the Manne *et al.* algorithm. As we use the exact same variables but the *end*-variable and because the matching is only defined on the common variables, the correctness follows from Manne *et al.* paper. Moreover, we can easily show that if C is stable in MaxMatch, then no rule from the Manne *et al.* algorithm but the *Update* rule can be performed in C. Unfortunately, it is not straightforward to prove that the *Update* rule from Manne *et al.* algorithm cannot be executed in C. Indeed, our *Update* rule is more difficult to execute than the one of Manne *et al.* in the sens that some possible *Update* in Manne *et al.* are not possible in our algorithm. By the way, this is why our algorithm has a better time complexity since the number of partially exploited augmented path destruction in our algorithm is smaller than in the Manne *et al.* algorithm. In particular, we have to prove that in a stable configuration, for any matched node, if $p_u \neq null$, then $end_{p_u} = True$. To prove that, we need Lemmas A, B, C, D, E and a part of the proof from Theorem 4.1. Observe that from these results, the correctness is straightforward without using the Manne *et al.* proof.

Let $Ask : V \to V \cup \{null\}$ be a function where $Ask(u) = AskFirst(u)$ if $AskFirst(u) \neq null$, otherwise $Ask(u) = AskSecond(u)$. We will say a node makes a *match* rule if it performs a *MatchFirst* or *MatchSecond* rule.

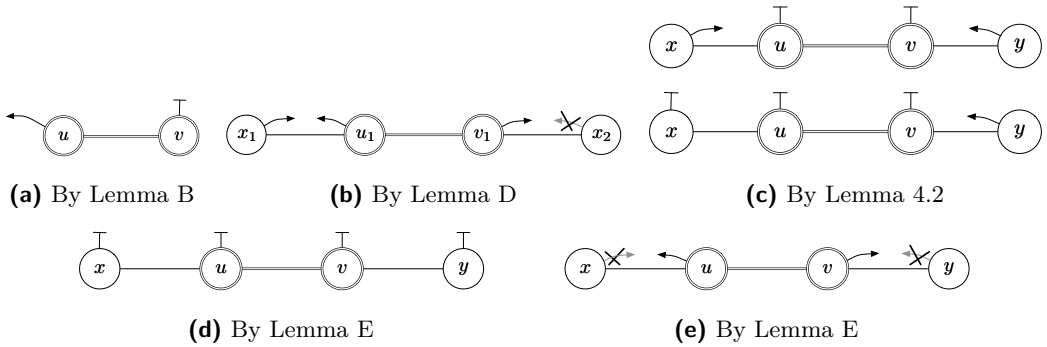
For the correctness part, we prove that in a stable configuration, $\mathcal{M}^+$ is a 2/3-approximation of a maximum matching on graph G. To do that we demonstrate there is no 3-augmenting path on $(G, \mathcal{M}^+)$. In particular we prove that for any edge $(u, v) \in \mathcal{M}$, we have either $p_u = p_v = null$, or u and v have two distinct single neighbors they are rematched with, *i.e.*, $\exists x \in single(N(u)), \exists y \in single(N(v))$ with $x \neq y$ such that $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$. In order to prove that, we show every other case for (u, v) is impossible. Main studied cases are shown in Figure 2. Finally, we prove that if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$.

▶ **Lemma A.** *In any stable configuration, we have the following properties:*

- $\forall u \in matched(V) : p_u = Ask(u)$;
- $\forall x \in single(V)$ *: if* $p_x = u$ *with* $u \neq null$*, then* $u \in matched(N(x)) \wedge p_u = x \wedge end_u = end_x$.

▶ **Lemma B.** *Let (u, v) be an edge in $\mathcal{M}$. Let C be a configuration. If $p_u \neq null \wedge p_v = null$ holds in C (see Fig. 2a), then C is not stable.*

▶ **Lemma C.** *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a stable configuration. In C, if $p_x = u$, $p_u = x$, $p_v = y$ and $p_y = u$, then $end_x = end_u = end_v = end_y = True$.*



(a) By Lemma B **(b)** By Lemma D **(c)** By Lemma 4.2

(d) By Lemma E **(e)** By Lemma E

Figure 2 3-augmenting paths that are not possible in a stable configuration.

▶ **Lemma D.** *Let (x_1, u_1, v_1, x_2) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_{x_1} = u_1 \wedge p_{u_1} = x_1 \wedge p_{v_1} = x_2 \wedge p_{x_2} \neq v_1$ holds in C (see Fig. 2b), then C is not stable.*

Proof. This is a sketch of the proof. We can prove that there exists a 3-augmenting path (x_2, u_2, v_2, x_3) such that $p_{x_2} = u_2 \wedge p_{u_2} = x_2 \wedge p_{v_2} = x_3 \wedge p_{x_3} \neq v_2$. This augmenting path has the exact same properties than the first considered augmenting path (x_1, u_1, v_1, x_2) and in particular u_1 is *First*. Now we can continue the construction in the same way. Therefore, for C to be stable, there must to exist a chain of 3-augmenting paths $(x_1, u_1, v_1, x_2, u_2, v_2, x_3, \ldots, x_i, u_i, v_i, x_{i+1}, \ldots)$ where $\forall i \geq 1 : (x_i, u_i, v_i, x_{i+1})$ is a 3-augmenting path with $p_{x_i} = u_i \wedge p_{u_i} = x_i \wedge p_{v_i} = x_{i+1} \wedge p_{x_{i+1}} = v_{i+1}$ and u_i is *First*. Thus, $x_1 < x_2 < \ldots < x_i < \ldots$ since the u_i will always be *First*. Since graph G is finite some x_k must be equal to some x_ℓ with $\ell \neq k$ which contradicts the fact that the identifier's sequence is strictly increasing. ◀

▶ **Lemma E.** *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $(p_u = x \wedge p_x \neq u \wedge p_v = y \wedge p_y \neq v)$ or if $(p_y = p_u = p_v = p_y = null)$ holds in C (see Fig. 2d and Fig. 2e respectively), then C is not stable.*

▶ **Theorem 4.1.** *In a stable configuration we have, $\forall (u, v) \in \mathcal{M}$:*

- $p_u = p_v = null$ *or*
- $\exists x \in single(N(u)), \exists y \in single(N(v))$ *with* $x \neq y$ *such that* $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$.

Proof. We will prove that all cases but these two are not possible in a stable configuration. First, Lemma B says the configuration cannot be stable if one of p_u or p_v is not *null*.

Second, assume that $p_u \neq null \wedge p_v \neq null$. Let $p_u = x$ and $p_v = y$. Observe that $x \in single(N(u))$ (resp. $y \in single(N(v))$), otherwise u (resp. v) is eligible for *Update*.

Case $x \neq y$: If $p_x \neq u$ and $p_y \neq v$ then Lemma E says the configuration cannot be stable. If $p_x = u$ and $p_y \neq v$ then Lemma D says the configuration cannot be stable. Thus, the only remaining possibility when $p_u \neq null$ and $p_v \neq null$ is: $p_x = u$ and $p_y = v$.

Case $x = y$: $Ask(u)$ (resp. $Ask(v) \neq null$), otherwise u (resp. v) is eligible for a *ResetMatch*. W.l.o.g. let us assume that u is *First*. $x = AskFirst(u)$ (resp. $x = AskSecond(v)$), otherwise u (resp. v) is eligible for *MatchFirst* (resp. *MatchSecond*). Thus $AskFirst(u) = AskSecond(v)$ which is impossible according to these two predicates. ◀

▶ **Lemma 4.2.** *Let x be a single node. In a stable configuration, if $p_x = u, u \neq null$ (see Fig. 2c) then it exists a 3-augmenting path (x, u, v, y) on $(G, \mathcal{M})$ such that $p_x = u \wedge p_u = x \wedge p_v = y \wedge p_y = v$.*

Proof. By Lemma A, if $p_x = u$ with $u \neq null$ then $u \in matched(N(x))$ and $p_u = x$. Since $p_u \neq null$, by Theorem 4.1 the result holds. ◀

Thus, in a stable configuration, for all edges $(u, v) \in \mathcal{M}$, if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$. In other words, we obtain:

▶ **Corollary 4.3.** *In a stable configuration, there is no 3-augmenting path on* $(G, \mathcal{M}^+)$ *left.*

5 Convergence Proof

This section is devoted to a sketch of the convergence proof. In the following, μ will denote the number of matched nodes and σ the number of single nodes.

The first step consists in proving that the values of s and *end* represent the different phases of the path exploitation. Recall that $s_u = True$ means $p_{p_u} = u$. Moreover $end_u = True$ means that the path is fully exploited. We can easily prove that after one activation of a matched node u, $s_u = True$ implies $p_{p_u} = u$ (Lemma F). However, a bad initialization of end_{m_u} to *True* can induce u to wrongly write *True* in end_u. But this can appear only once and thus, the second times u writes *True* in end_u means that a 3-augmenting path involving u has been fully exploited (Theorem 5.1).

▶ **Lemma F.** *Let* u *be a matched node. Consider an execution* $\mathcal{E}$ *starting after* u *executed some rule. Let* C *be any configuration in* $\mathcal{E}$*. In* C*, if* $s_u = True$ *then* $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$*.*

▶ **Theorem 5.1.** *In any execution, a matched node* u *can write* $end_u := True$ *at most twice.*

We now count the number of destruction of partially exploited augmenting paths. Recall that in Manne *et al.* algorithm, for one fully exploited augmenting path, it is possible to destroy a sub-exponential number of partially exploited ones.

In our algorithm, observe that for a path destruction, the set of single neighbors that are candidates for a matched edge has to change and this change can only occur when a single node changes its *end*-value. Such a change induces a path destruction if a matched node takes into account this modification by applying an *Update* rule. So, we first count the number of time a single node can change its *end*-value (Lemma G) and then we deduce the number of time a matched node can execute *Update* (Corollary H). Finally, we conclude we destroy at most $O(n^2)(= O(\Delta(\sigma + \mu)))$ partially exploited augmenting path.

The rest of the proof consists in counting the number of moves that can be performed between two *Update*, allowing us to conclude the proof (Theorem I).

In the following, we detailed point by point the idea behind each result cited above.

Since single nodes just follow orders from their neighboring matched nodes, we can count the number of times single nodes can change the value of their *end* variable. There are σ possible modifications due to bad initializations. A matched node u can write *True* twice in end_u, so end_u can be *True* during 3 distinct sub-executions. As a single node x copies the *end*-value of the matched node it points to ($p_x = u$), then a single node can change its *end*-value at most 3 times as well. And we obtain 6μ modifications.

▶ **Lemma G.** *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most* $\sigma + 6\mu$ *times.*

We count the maximal number of *Update* rule that can be performed in any execution. To do that, we observe that the first line of the *Update* guard can be *True* at most once in an execution (Lemma L). Then we prove for the second line of the guard to be *True*, a single node has to change its *end* value. Thus, for each single node modification of the *end*−value, at most all matched neighbors of this single node can perform an *Update* rule.

▶ **Corollary H.** *Matched nodes can execute at most* $\Delta(\sigma + 6\mu) + \mu$ *times the* Update *rule.*

Third, we count the maximal number of moves performed by matched nodes between two *Update*. The idea is that in an execution without *Update*, α and β values of all matched nodes remain constant. Thus, in these small executions, at most one augmenting path is detected per matched edge and at most one rematch attempt is performed per matched edge. We obtain that the maximal number of moves of a matched node in these small executions is 12. By the previous remark and Corollary H, we obtain:

▶ **Theorem I.** *In any execution, matched nodes can execute at most* $12\mu(\Delta(\sigma + 6\mu) + \mu)$ *rules.*

Finally, we count the maximal number of moves that single nodes can perform, counting rule by rule. The *ResetEnd* is done at most once. The number of *UpdateEnd* is bounded by the number of times single nodes can change their *end*-value, so it is at most $\sigma + 6\mu$. Finally, *UpdateP* is counted as follows: between two consecutive *UpdateP* executed by a single node x, a matched node has to make a move. The total number of executed *UpdateP* is then at most $12\mu(\Delta(\sigma + 6\mu) + \mu) + 1$.

▶ **Corollary J.** *The algorithm* MaxMatch *converges in* $O(n^3)$ *steps under the adversarial distributed daemon.*

Due to the lack of space, we cannot give the whole convergence proof. We choose to present the proof of Theorem 5.1 since it is the key point of the convergence proof. Indeed, with this result, we have a first strong step leading to the proof of the silent property of our algorithm. The remaining of this section is devoted to prove Theorem 5.1. The rest of the proof is placed in [4].

5.1 A matched node can write *True* in its *end*-variable at most twice

The first two lemmas are technical lemmas.

▶ **Lemma K.** *Let* u *be a matched node. Consider an execution* $\mathcal{E}$ *starting after* u *executed some rule. Let* C *be any configuration in* $\mathcal{E}$*. If* $end_u =$*True in* C *then* $s_u =$*True as well.*

▶ **Lemma L.** *Let* u *be a matched node and* $\mathcal{E}$ *be an execution containing a transition* $C_0 \mapsto C_1$ *where* u *makes a move. From* C_1*, the predicate in the first line of the guard of the* Update *rule will never hold from* C_1*.*

Now, we will focus on particular configurations for a matched edge (u, v) corresponding to the fact they have completely exploited a 3-augmenting path.

▶ **Lemma 5.1.** *Let* (u, v) *be a matched edge,* $\mathcal{E}$ *be an execution and* C *be a configuration of* $\mathcal{E}$*. If in* C*, we have:*
1. $p_u \in single(N(u)) \wedge p_u = AskFirst(u) \wedge p_{p_u} = u$*;*
2. $p_v \in single(N(v)) \wedge p_v = AskSecond(v) \wedge p_{p_v} = v$*;*
3. $s_u = end_u = s_v = end_v =$*True;*

then neither u *nor* v *will ever be eligible for any rule from* C*.*

Proof. Observe first that neither u nor v are eligible for any rule in C. Moreover, p_u (resp. p_v) is not eligible for an *UpdateP* move since u (resp. v) does not make any move. Thus p_{p_u} and p_{p_v} will remain constant since u and v do not make any move and so neither u nor v will ever be eligible for any rule from C. ◀

The configuration C described in Lemma 5.1 is called a configuration $stop_{uv}$. From such a configuration neither u nor v will ever be eligible for any rule.

In Lemmas 5.4 and M, we consider executions where a matched node u writes *True* in end_u twice, and we focus on the transition $C_0 \mapsto C_1$ where u performs its second writing. Lemma 5.4 shows that, if u is *First* in C_0, then C_1 is a $stop_{um_u}$ configuration. Lemma M shows that, if u is *Second* in C_0, then either C_1 is a $stop_{um_u}$ configuration or it exists a configuration C_3 such that $C_3 > C_1$ and u does not make any move from C_1 to C_3 and C_3 is a $stop_{um_u}$ configuration.

Lemma 5.2 and Corollary 5.3 are required in order to prove Lemmas 5.4 and M.

▶ **Lemma 5.2.** *Let (u, v) be a matched edge. Let $\mathcal{E}$ be some execution in which v does not execute any rule. If it exists a transition $C_0 \mapsto C_1$ in $\mathcal{E}$ where u writes True in end_u, then u is not eligible for any rule from C_1.*

Proof. To write *True* in end_u in transition $C_0 \mapsto C_1$, u must have executed a *match* rule. According to this rule, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds C_0 with $p_u \in single(N(u))$, otherwise u would have executed an *Update* instead of a *match* rule. Now, in $C_0 \mapsto C_1$, p_u cannot execute *UpdateP* then it cannot change its p-value and v does not execute any move then it cannot change $Ask(u)$. Thus, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds in both C_0 and C_1.

Assume now by contradiction that u executes a rule after configuration C_1. Let $C_2 \mapsto C_3$ be the next transition in which it executes a rule. Recall that between configurations C_1 and C_2 both u and v do not execute rules. Observe also that p_u is not eligible for *UpdateP* between these configurations. Thus $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds from C_0 to C_2. Moreover the following points hold as well between C_0 and C_2 since in $C_0 \mapsto C_1$ u executed a *match* rule and v does not apply rules in $\mathcal{E}$:

- α_u, α_v, β_u and β_v do not change.
- The values of the variables of v do not change.
- $Ask(u)$ and $Ask(v)$ do not change.
- If u was *First* in C_0 it is *First* in C_2 and the same holds if it was *Second*.

Using these remarks, we start by proving that u is not eligible for *ResetMatch* in C_2. If it is *First* in C_2, this holds since $AskFirst(u) \neq null$ and $AskSecond(u) = null$. If it is *Second* then to be eligible for *ResetMatch*, $s_v =$*False* must hold in C_2 since $AskSecond(u) \neq null$. Since u executed $end_u =$*True* in $C_0 \mapsto C_1$ and since u was *Second* in C_0, then necessarily $s_v =$*True* in C_0 and thus in C_2 (using remark 2 above). So u is not eligible for *ResetMatch* in C_2.

We show now that u is not eligible for an *Update* in C_2. The α and β variables of u and v remain constant between C_0 and C_2. Thus if any of the three first disjunctions in the *Update* rule holds in C_2 then it also holds in C_0 and in $C_0 \mapsto C_1$ u should have executed an *Update* since it has higher priority than the *match* rules. Moreover since in C_2 $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds, the last two disjunctions of *Update* are *False* and we can state u is not eligible for this rule.

We conclude the proof by showing that u is not eligible for a *match* rule in C_2. If u was *First* in C_0 then it is *First* in C_2. To write *True* in end_u then $(p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})$ must hold in C_0. Since in $C_0 \mapsto C_1$ v does not

execute rules, it also holds in C_1. The same remark between configurations C_1 and C_2 implies that this predicate holds in C_2. Thus in C_2, all the three conditions of the *MatchFirst* guard are *False* and u not eligible for *MatchFirst*. A similar remark if u is *Second* implies that u will not be eligible for *MatchSecond* in C_2 if it was *Second* in C_0. ◀

▶ **Corollary 5.3.** *Let (u, v) be a matched edge. In any execution, if u writes True in end_u twice, then v executes a rule between these two writing.*

▶ **Lemma 5.4.** *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes True in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes True in end_u for the second time in $\mathcal{E}$. If u is* First *in C_0 then the following holds:*

1. *in configuration C_0,*
   - **(a)** $s_v = end_v = True$;
   - **(b)** $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u = True \wedge p_v = AskSecond(v)$;
   - **(c)** $p_u \in single(N(u))$;
   - **(d)** $p_v \in single(N(v)) \wedge p_{p_v} = v$;
2. *v does not execute any move in $C_0 \mapsto C_1$;*
3. *in configuration C_1,*
   - **(a)** $s_u = end_u = True$;
   - **(b)** $p_u \in single(N(u)) \wedge p_v \in single(N(v))$;
   - **(c)** $s_v = end_v = True$;
   - **(d)** $p_u = AskFirst(u) \wedge p_v = AskSecond(v)$;
   - **(e)** $p_{p_u} = u \wedge p_{p_v} = v$.

Proof. We prove Point 1a. Observe that for u to write *True* in end_u, end_v must be *True* in C_0. By Lemma K this implies that s_v is *True* as well. Now Point 1b holds by definition of the *MatchFirst* rule. As in C_0, u already executed an action, then according to Lemma L, Point 1c holds and will always hold. By Corollary 5.3, u cannot write *True* consecutively if v does not execute moves. Thus at some point before C_0, v applied some rule. This implies that in configuration C_0, since $s_v = True$, by Lemma F, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$. Thus Point 1d holds.

We now show that v does not execute any move in $C_0 \mapsto C_1$ (Point 2). Recall that v already executed an action before C_0, so by Lemma L, line 1 of the *Update* guard does not hold in C_0. Moreover, by Point 1d, line 2 does not hold either. Thus, v is not eligible for *Update* in C_0. We also have that $s_u = True$ and $AskSecond(v) \neq null$ in C_0, thus v is not eligible for *ResetMatch*. Observe now that by Points 1a, 1b and 1d, v is not eligible for *MatchSecond* in C_0. Finally v cannot execute *MatchFirst* since $AskFirst(v) = null$. Thus v does not execute any move in $C_0 \mapsto C_1$ and so Point 2 holds.

In C_1, end_u is *True* by hypothesis and according to Point 1b, u writes *True* in s_u in transition $C_0 \mapsto C_1$. Thus Point 3a holds. Points 3b holds by Points 1c and 1d. Points 3c holds by Points 1a and 2. $AskFirst(u)$ and $AskSecond(v)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v executes an *Update* in this transition. Moreover p_v remains constant in $C_0 \mapsto C_1$ by Point 2 and p_u remains constant also since it writes $AskFirst(u)$ in p_u in this transition while $p_u = AskFirst(u)$ in C_0. Thus Points 3d holds. Observe that nor p_u neither p_v is eligible for an *UpdateP* in C_0, thus Point 3e holds. ◀

Now, we consider the case where u is *Second*.

▶ **Lemma M.** *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes True in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes True in end_u for the second time in $\mathcal{E}$. If u is* Second *in C_0 then the following holds:*

1. *in configuration* C_0*:*
   (a) $s_v = True \wedge p_v = AskFirst(v)$ *and*
   (b) $p_v \in single(N(v)) \wedge p_{p_v} = v$
2. *in transition* $C_0 \mapsto C_1$*,* v *is not eligible for* Update *nor* ResetMatch*;*
3. *in configuration* C_1*,*
   (a) $s_u = end_u = True$ *and*
   (b) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$ *and*
   (c) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$ *and*
   (d) $s_v = True$*;*
4. u *is not eligible for any move in* C_1*;*
5. *If* $end_u = False$ *in* C_1 *then the following holds:*
   (a) *From* C_1*,* v *executes a next move and this move is a* MatchFirst*;*
   (b) *Let us assume this move (the first move of* v *from* C_1*) is done in transition* $C_2 \mapsto C_3$*. In configuration* C_3*, we have:*
      (i) $s_u = end_u = True$ *and*
      (ii) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$ *and*
      (iii) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$ *and*
      (iv) $s_v = True$ *and*
      (v) u *does not execute moves between* C_1 *and* C_3 *and*
      (vi) $end_v = True$*.*

▶ **Theorem 5.1.** *In any execution, a matched node* u *can write* $end_u := True$ *at most twice.*

Proof. Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes *True* in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes *True* in end_u for the second time in $\mathcal{E}$. If u is *First* (resp. *Second*) in C_0 then from Lemmas 5.1 and 5.4, (resp. Lemma M), from C1, neither u nor v will ever be eligible for any rule. ◀

References

1 Y. Asada and M. Inoue. An efficient silent self-stabilizing algorithm for 1-maximal matching in anonymous networks. In *WALCOM: Algorithms and Computation – 9th International Workshop*, pages 187–198. Springer International Publishing, 2015.

2 P. Berenbrink, T. Friedetzky, and R. A. Martin. On the stability of dynamic diffusion load balancing. *Algorithmica*, 50(3):329–350, 2008. doi:10.1007/s00453-007-9081-y.

3 J. Cohen, K. Maamra, G. Manoussakis, and L. Pilard. The Manne *et al.* self-stabilizing 2/3-approximation matching algorithm is sub-exponential. *CoRR*, abs/1604.08066, 2016. URL: http://arxiv.org/abs/1604.08066.

4 J. Cohen, K. Maamra, G. Manoussakis, and L. Pilard. Polynomial self-stabilizing algorithm and proof for a 2/3-approximation of a maximum matching. *CoRR*, abs/1611.06038, 2016. URL: http://arxiv.org/abs/1611.06038.

5 S. Dolev. *Self-Stabilization.* MIT Press, 2000.

6 D. E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003.

7 B. Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996. doi:10.1006/jcss.1996.0075.

8 N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *J. Parallel Distrib. Comput.*, 70(4):406–415, 2010.

9 S. T. Hedetniemi, D. Pokrass Jacobs, and P. K. Srimani. Maximal matching stabilizes in time o(m). *Inf. Process. Lett.*, 80(5):221–223, 2001.

10 M. Hoefer. Local matching dynamics in social networks. *Inf. Comput.*, 222:20–35, 2013.

11 J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

12 S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992.

13 D. Knuth. *Marriages stables et leurs relations avec d'autres problèmes combinatoires*. Les Presses de l'Université de Montréal, 1976.

14 F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *9th Int. Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, Lecture Notes in Computer Science, pages 383–393. Springer, 2007.

15 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science (TCS)*, 410(14):1336–1345, 2009.

16 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science (TCS)*, 412(40):5515–5526, 2011.

17 R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, pages 259–269. Springer, 1999.

18 M. Touati, R. El-Azouzi, M. Coupechoux, E. Altman, and J. M. Kelif. Controlled matching game for user association and resource allocation in multi-rate wlans? In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 372–380, Sept 2015. doi:10.1109/ALLERTON.2015.7447028.

19 V. Turau and B. Hauck. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theoretical Computer Science (TCS)*, 412(40):5527–5540, 2011.

Distributed Stable Matching with Similar Preference Lists

Pankaj Khanchandani[1] and Roger Wattenhofer[2]

1 **ETH Zürich, Zürich, Switzerland**
`kpankaj@ethz.ch`
2 **ETH Zürich, Zürich, Switzerland**
`wattenhofer@ethz.ch`

Abstract

Consider a complete bipartite graph of $2n$ nodes with n nodes on each side. In a round, each node can either send at most one message to a neighbor or receive at most one message from a neighbor. Each node has a preference list that ranks all its neighbors in a strict order from 1 to n. We introduce a non-negative similarity parameter $\Delta < n$ for the preference lists of nodes on one side only. For $\Delta = 0$, these preference lists are same and for $\Delta = n - 1$, they can be completely arbitrary. There is no restriction on the preference lists of the other side. We show that each node can compute its partner in a stable matching by receiving $O(n(\Delta + 1))$ messages of size $O(\log n)$ each. We also show that this is optimal (up to a logarithmic factor) if Δ is constant.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases distributed stable matching, similar preference lists, stable matching, stable marriage

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.12

1 Introduction

In 1962, Gale and Shapley proposed an algorithm [5] to find a *stable matching* between a set of n men and n women, where each participant has a preference list ranking all the participants of opposite gender in a strict order. The algorithm computes a match for each man (and each woman) so that any pair of man and woman who are not matched do not both prefer each other to their current matches. The Gale-Shapley algorithm is one of the most influential results in 20^{th} century science, and forms the basis of the 2012 Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel.

The Gale-Shapley algorithm is also one of the very first distributed algorithms, as all the unmatched men can send their proposals simultaneously. Despite its distributed design, there are worst-case problem instances which basically execute sequentially as there are $\Theta(n^2)$ steps with only a single unmatched man [8]. These worst-case problem instances however feel very contrived, as participants need completely divergent preferences. One might expect that real world preference lists will be somewhat similar.

In this paper, we want to know whether similar preference lists allow for designing faster distributed algorithms. Towards this, we introduce a parameter Δ which is the smallest number so that the difference between the maximum and minimum ranks assigned to any man is at most Δ. A smaller value of Δ implies that the difference in the positions of a man in the preference lists of any two women is also smaller and the preference lists of women are relatively similar. The preference lists of men are still allowed to be arbitrary, independent of Δ.

Each participant is represented by a node in our distributed model and computation proceeds in synchronous rounds. If we use the traditional message passing model of distributed computing, a man can send his proposals to all the n women simultaneously, where n can be arbitrarily large. This does not seem to be a realistic social construct. Instead, in this paper we use a communication model where each node can either send *or* receive at most one message in a single round. It is an exciting open question whether a node can compute its stable partner by receiving $o(n^2)$ values or messages of size $O(\log n)$ each.

We give an algorithm so that any node computes its partner in a stable matching by receiving $O(n(\Delta + 1))$ messages of size $O(\log n)$ each. We also show that any node requires $\Omega(n/\log n)$ messages to compute its partner. Thus, our algorithm is nearly optimal for constant Δ.

2 Related Work

The seminal paper by Gale and Shapley [5] has inspired lots of research on stable matching. The book of Gusfield and Irving [6] extensively survey the problem and its many variants. Irving et al. give a version where the preference lists of one set of participants (say women) are derived from a master preference list so that the men present in the master list occur in the preference lists of women in the same order as in the master list [7]. This results in similar preference lists but the order of a group of men remains same across all the women. The similarity measure Δ that we use does not enforce a strict order on any particular group of men. To the best of our knowledge, this measure has not been discussed before.

Distributed stable matching with a goal of keeping the preference lists private to an individual has also been studied [2, 3]. In our work, our focus is to solve the problem rather collaboratively, minimizing the information required by each node to compute its stable partner. Kipnis and Patt-Shamir consider a distributed version in which a man is not required to rank all the women and give a lower bound on the information required by a node using an incomplete bipartite graph [9]. To understand the hardness of the problem when topology is not a limiting factor, we consider that each man ranks all the women and vice-versa or a complete bipartite graph.

Amira et al. consider a variant where a complete bipartite graph has weighted edges [1]. The preference list of a node is then derived according the order of weights of edges that are incident at the node. They give an algorithm in which any node can compute its partner using only $O(n\sqrt{n})$ values. For the problem instances that they consider, the preference lists of women is dependent on the preference lists of men or the other way around. In our case, the preference lists of men can be completely arbitrary and that of women can be chosen independently, only subject to the parameter Δ. Also, the values required by a node to compute a stable matching can be larger or smaller depending on the parameter Δ.

In parallel algorithms with $\Theta(n)$ processes, a process might still need to access $\Omega(n^2)$ values before a stable partner is computed [12, 13]. Distributed algorithms for approximate stable matching have also been studied. For example, Floréen et al. consider the case where the preference lists have bounded length and give a local algorithm for a stable matching with only few unstable edges [4]. Ostrovsky et al. give an algorithm for almost stable matching that terminates in poly-logarithmic rounds for arbitrary preferences [11].

3 Model

Let $\mathbb{M} = \{m_1, m_2, \ldots, m_n\}$ be the set of men and $\mathbb{W} = \{w_1, w_2, \ldots, w_n\}$ be the set of women. In a *matching*, any man is matched to at most one woman and any woman is matched to at

$$M = \begin{pmatrix} w_1 & w_2 & w_3 & w_4 & w_5 \\ w_1 & w_4 & w_5 & w_3 & w_2 \\ w_4 & w_3 & w_1 & w_5 & w_2 \\ w_5 & w_4 & w_1 & w_2 & w_3 \\ w_5 & w_1 & w_2 & w_3 & w_4 \end{pmatrix} \qquad W = \begin{pmatrix} m_1 & m_2 & m_3 & m_4 & m_5 \\ m_1 & m_2 & m_3 & m_4 & m_5 \\ m_1 & m_2 & m_3 & m_4 & m_5 \\ m_1 & m_2 & m_3 & m_4 & m_5 \\ m_1 & m_2 & m_3 & m_4 & m_5 \end{pmatrix}$$

Figure 1 Sample Preference Matrices for $n = 5$, $\Delta = 0$.

most one man. A *perfect matching* is a matching where all the men and women are matched. Given a perfect matching, a pair of man and woman that are not matched to each other is called a *blocking pair* if they prefer each other to their matched partners. Thus, a stable matching is a perfect matching without a blocking pair.

We denote the preference lists of men by matrix M where an entry M_{ij} is the j^{th} ranked woman by man m_i. Similarly, matrix W is so that an entry W_{ij} is the j^{th} ranked man by woman w_i. Thus, a woman w_i prefers man W_{ij} to the man W_{ik} if $j < k$ and a man m_i prefers woman M_{ij} to the woman M_{ik} if $j < k$. The similarity parameter Δ is the smallest value so that the following holds for the matrix W.

$$W_{ij} = W_{pq} \Rightarrow |j - q| \leq \Delta \tag{1}$$

Thus, the range in which a man is ranked by all the women is at most Δ large.

In the distributed setting, the set $\mathbb{M}$ and $\mathbb{W}$ are the nodes and the network is a complete bipartite graph with $\mathbb{M}$ and $\mathbb{W}$ on the opposite sides. The algorithm proceeds in sequence of synchronous rounds. In each round, a node can either

1. Send a message to at most one neighbor and do local computations; or
2. Receive a message from at most one neighbor and do local computations.

If several messages are sent to the same node in the same round, then none are received by the intended recipient. The messages are of size $O(\log n)$.

Let us first see that any algorithm for finding the stable matching in this model requires $\Omega(n/\log n)$ rounds when $\Delta = 0$. Afterwards, we will see a sequential algorithm and its analysis which we later develop into the distributed algorithm. Finally, we show its correctness.

4 Lower Bound

Figure 1 shows sample matrices M and W for $n = 5$ and $\Delta = 0$. As $\Delta = 0$, the preference lists of all the women are same. For simplicity, we assume that the man m_i is ranked i^{th} by all women.

The man m_1 is ranked first by all the women. Thus, m_1 must be matched to his first preference, w_1, in any stable matching. Otherwise, m_1 and w_1 would form a blocking pair.

The man m_2 is ranked second by all the women and his most preferred woman is w_1. However, m_2 cannot be matched to w_1 as she must be matched to m_1. On the other hand, m_2 must be matched to w_4, which is his most preferred woman excluding w_1. Otherwise, m_2 and w_4 would form a blocking pair. We can generalize this into the following lemma.

▶ **Lemma 1.** *Consider the stable matching problem with n men and n women where any entry $W_{ij} = m_j$, i.e., $\Delta = 0$. In any stable matching, a man m_j is matched to his most preferred woman excluding the women matched to men m_i, $i < j$.*

Proof. Assume a stable matching S in which a j exists so that m_j is *not* matched to his most preferred women when the women matched to m_i, $i < j$ are excluded.

$$M = \begin{pmatrix} w_1 & w_5 & w_4 & w_3 & w_2 \\ w_4 & w_3 & w_1 & w_5 & w_2 \\ w_1 & w_4 & w_5 & w_3 & w_2 \\ w_5 & w_4 & w_2 & w_1 & w_3 \\ w_1 & w_2 & w_3 & w_4 & w_5 \end{pmatrix} \qquad W = \begin{pmatrix} m_5 & m_3 & m_2 & m_4 & m_1 \\ m_3 & m_2 & m_5 & m_1 & m_4 \\ m_3 & m_5 & m_4 & m_2 & m_1 \\ m_5 & m_2 & m_3 & m_4 & m_1 \\ m_5 & m_3 & m_2 & m_1 & m_4 \end{pmatrix}$$

Figure 2 Sample Preference Matrices for $n = 5$, $\Delta = 2$.

Say, that m_j is matched to a woman w and w^* is his most preferred woman excluding the women matched to m_i, $i < j$. Clearly, m_j prefers w^* to w. Consider the woman w^*. She is not matched to m_j by our assumption. Also, she is not matched to m_i for $i < j$ either. Thus, w^* prefers m_j to her match m_k, $k > j$. Hence, m_j and w^* form a blocking pair and S is not a stable matching. ◀

Thus, m_j should know the women matched to men m_i, $i < j$ to determine his partner. We exploit this to show a lower bound on the number of rounds necessary to compute the stable matching as follows. We will use the notation $[1, n]$ for the set of integers between 1 and n inclusive.

▶ **Lemma 2.** *It takes $\Omega(n/\log n)$ rounds to compute a stable matching for n men and n women when $W_{ij} = m_j$, i.e., $\Delta = 0$.*

Proof. Consider a man m_k, $k \in [1, n]$. As a corollary of Lemma 1, m_k is matched to a woman M_{kj}, $j \leq k$. Let B be the set of $k - 1$ highest ranked women by m_k, i.e., $B = \{M_{kj} : j \in [1, k-1]\}$. Let A be the set of women matched to the men ranked higher than m_k by the women, i.e., $A = \{w_i : w_i \text{ matched to } m_j,\ j < k\}$. Using Lemma 1, $A = B$ if m_k is matched to M_{kk}. On the other hand, if m_k is not matched to M_{kk}, then m_k is matched to a woman in set B that is not in set A and $A \neq B$.

It is known that $\Omega(n)$ bits must be exchanged between two parties, if each of them is given a subset of $[1, n]$ and they want to determine if the subsets are same [10] [1]. Here, node m_k having set B determined if it was equal to set A present in the other nodes. Thus, $\Omega(n)$ bits must be exchanged with m_k. As the message size is $O(\log n)$, the algorithm must run for $\Omega(n/\log n)$ rounds. ◀

Let us now see a sequential algorithm in which any man needs to propose $O(\Delta)$ times to find the stable matching. Later, we will implement this algorithm in our distributed model.

5 Sequential Algorithm

We will use the notation $c(m_j)$ to denote the smallest column number of W in which the man m_j occurs. We show sample input matrices M and W for $n = 5$ and $\Delta = 2$ in Figure 2. For this input, $c(m_5) = c(m_3) = 1$, $c(m_2) = 2$, $c(m_4) = 3$ and $c(m_1) = 4$. Also, we define a bijective function $l : [1, n] \to [1, n]$ so that $c(m_j) < c(m_k) \Rightarrow l(j) < l(k)$. As the function l is bijective, l^{-1} is defined as well. For example, following is an assignment of l and l^{-1} given the preference matrix W in Figure 2: $l(5) = 1$, $l(3) = 2$, $l(2) = 3$, $l(4) = 4$, $l(1) = 5$ and $l^{-1}(1) = 5$, $l^{-1}(2) = 3$, $l^{-1}(3) = 2$, $l^{-1}(4) = 4$, $l^{-1}(5) = 1$.

[1] The result also applies if the sizes of the subsets are same as in our case. As otherwise, one could exchange the sizes in $O(\log n)$ bits and compute the result using $o(n)$ bits.

Algorithm 1: Sequential Stable Matching: The loop at Line 3 ends after all men are matched. If a woman w receives her first proposal (Line 6), then w is deleted from the preference list of all men m_j where $c(m_j) > i + \Delta$ (loop at Line 14).

```
1  F ← 𝕄;
2  i ← 1;
3  while F ≠ ∅ do
4      m ← m_j where j = l^{-1}(min{l(k) : m_k ∈ F});
5      i = max{i, c(m)};
6      m proposes to the next available (not already deleted) woman w from his
        preference list;
7      if w is enaged and does not prefer m to her current partner then
8          w rejects m's proposal;
9      else
10         if w is engaged to a less preferred partner m' then
11             w disengages with m';
12             F ← F ∪ {m'};
13         else // w is not engaged
14             forall j ∈ [1, n] do
15                 if c(m_j) > i + Δ then
16                     Delete w from m_j's preference list;
17         w accepts m;
18         F ← F\{m};
```

Algorithm 1 describes the sequential stable matching algorithm. The set F contains men that are free and is initialized with all the men. The variable i is the maximum column number so that there is a man m_j with $c(m_j) = i$ that already proposed to someone. The free man that has the smallest value assigned by function l proposes the next women available in his list. Not all are available as some of them are deleted from his list during the algorithm. A woman accepts a proposal if she is free or if the proposal is better than her current partner. If it was the first proposal she accepted, then she is deleted from the preference lists of all the men m_j that have $c(m_j) > i + \Delta$.

We show an execution of the algorithm in Table 1. It can be checked that the final matching is indeed a stable matching. Also note that no more than three proposals $(\Delta + 1)$ are required for any man m_j. Let us now analyze the algorithm to see if we can generalize these observations.

6 Analysis of the Sequential Algorithm

It is rather easy to show that the algorithm always maintains a matching, i.e., each node is matched to at most one neighbor.

▶ **Lemma 3.** *Algorithm 1 always maintains a matching.*

Proof. A woman is never matched to more than one man as whenever she accepts a proposal, she always disengages with her current partner (if any). A man sends a proposal only if he is in set F. Initially, F contains all the men which are free. Whenever a man is accepted, he is

Table 1 An execution of Algorithm 1 on preference matrices of Figure 2. We use the following definition of function l: $l(5) = 1$, $l(3) = 2$, $l(2) = 3$, $l(4) = 4$ and $l(1) = 5$. Thus, we choose a free man in Line 4 that occurs earliest in the following list: m_5, m_3, m_2, m_4, m_1. We show the change in variables during the subsequent iterations of the loop at Line 3. The column 'F' shows the value of the variable F at the start of the iteration. The column 'Propose' is the proposal made during the iteration. The column i shows the value of variable i just before entries from preference lists are deleted. The column 'Delete' shows entries of M deleted during the iteration. The column 'Matching' is the matching at the end of the iteration.

F	Propose	i	Delete	Matching
$\mathbb{M}$	$m_5 \to w_1$	1	M_{11}	$\{(m_5, w_1)\}$
$\mathbb{M}\backslash m_5$	$m_3 \to w_1$	1		$\{(m_5, w_1)\}$
$\mathbb{M}\backslash m_5$	$m_3 \to w_4$	1	M_{13}	$\{(m_5, w_1), (m_3, w_4)\}$
$\mathbb{M}\backslash\{m_5, m_3\}$	$m_2 \to w_4$	2		$\{(m_5, w_1), (m_2, w_4)\}$
$\mathbb{M}\backslash\{m_5, m_2\}$	$m_3 \to w_5$	2		$\{(m_5, w_1), (m_2, w_4), (m_3, w_5)\}$
$\{m_1, m_4\}$	$m_4 \to w_5$	3		$\{(m_5, w_1), (m_2, w_4), (m_3, w_5)\}$
$\{m_1, m_4\}$	$m_4 \to w_4$	3		$\{(m_5, w_1), (m_2, w_4), (m_3, w_5)\}$
$\{m_1, m_4\}$	$m_4 \to w_2$	3		$\{(m_5, w_1), (m_2, w_4), (m_3, w_5), (m_4, w_2)\}$
$\{m_1\}$	$m_1 \to w_5$	4		$\{(m_5, w_1), (m_2, w_4), (m_3, w_5), (m_4, w_2)\}$
$\{m_1\}$	$m_1 \to w_3$	4		$\{(m_5, w_1), (m_2, w_4), (m_3, w_5), (m_4, w_2), (m_1, w_3)\}$

removed from F and whenever a man is disengaged, he is added to F. Thus, a matched man never sends a proposal and a man is matched to at most one woman. ◀

However, it is not clear that the algorithm computes a perfect matching as the loop of Line 3 may not terminate. We show that this does not happen using contradiction as follows.

▶ **Lemma 4.** *Algorithm 1 computes a perfect matching.*

Proof. If the algorithm terminates, then none of the men are free. Using Lemma 3 the statement holds. Thus, we now only need to show that the algorithm terminates.

Assume that the algorithm does not terminate. Then, there is a man m that has exhausted his preference list and is unmatched. Now, there are two possibilities for each woman w in m's preference list. First: m proposed to w and got rejected or disengaged later. Second: m did not propose to w as she was already deleted from m's preference list. In either case, w would still be matched as w never rejects a proposal if she is free and only accepts better proposals if already matched. This means that n women are matched to at most $n-1$ other men (excluding m). Using Lemma 3, this is a contradiction. ◀

And now that we have a perfect matching upon termination, we can assume a blocking pair in that perfect matching and show that such a pair does not exist. We introduce the notation $W_{[a,b][x,y]}$ to denote the set of elements in the sub-matrix of W formed between rows a, b inclusive and columns x, y inclusive. We will also use the shorthand $*$ to denote the range $[1, n]$ and a to denote the single element range $[a, a]$.

▶ **Lemma 5.** *The matching computed by Algorithm 1 is a stable matching.*

Proof. We know from Lemma 4 that the algorithm terminates in a perfect matching. Assume that m and w' form a blocking pair and w' prefers m to her final match m'.

Now, either m did or did not propose to w'. If m proposed to w', then it got rejected or disengaged later and w' prefers m' to m, a contradiction. If m did not propose to w', then

w' was removed from m's preference list. Say, w' was matched to m'' and $i = p$ when she was removed. Then, we conclude that $c(m) > p + \Delta$. Also, $c(m'') \leq p$ as m'' is matched to w' when $i = p$. Consequently, $m'' \notin W_{*[p+\Delta+1,n]}$ due to (1). Thus, w' prefers m'' to m. As w' only accepts better proposals in the future and m'' was her first match, she also prefers m' to m, a contradiction. ◀

There is an interesting property of this algorithm, namely, no man proposes more than $3\Delta + 1$ times. We will first show a bound on the number of men up to a certain number of columns of W. Later, we use this bound to show this property.

▶ **Lemma 6.** *The total number of men in $W_{*[1,j]}$ is at least j and at most $j + \Delta$.*

Proof. The elements of any given row $i \in [1, n]$ of W are all different and thus they are different in a given range $[1, j]$. Thus, we have $|W_{i[1,j]}| = j$ and $|W_{*[1,j]}| \geq j$.

If a man $m \in W_{*[1,j]}$, then (1) implies that $m \notin W_{*[j+\Delta+1,n]}$. As m occurs in every row of W, $m \in W_{i[1,j+\Delta]}$ for $i \in [1, n]$. Thus, $W_{*[1,j]} \subseteq W_{i[1,j+\Delta]}$ for $i \in [1, n]$ and we have $|W_{*[1,j]}| \leq (j + \Delta) - 1 + 1 = j + \Delta$. ◀

Now, we can use the above lemma to bound the number of proposals that are made by a man.

▶ **Lemma 7.** *The maximum number of proposals made by a man in Algorithm 1 is $3\Delta + 1$.*

Proof. Let m be some man and say that $c(m) = p$. Consider the first time t when all the men m_j with $c(m_j) \leq p - \Delta - 1$ are matched. When $i > p - \Delta - 1$ for the first time, then all the men m_j with $c(m_j) \leq p - \Delta - 1$ are already matched. Thus, $i \leq p - \Delta - 1$ at time t. Note that the set of men m_j with $c(m_j) \leq p - \Delta - 1$ is same as $W_{*[1,p-\Delta-1]}$. Using Lemma 6, this set is at least $p - \Delta - 1$ in size. Thus, at least $p - \Delta - 1$ free women received proposals when $i \leq p - \Delta - 1$. As $c(m) = p > (p - \Delta - 1) + \Delta$, at least $p - \Delta - 1$ women were deleted from m's preference list.

At time t, let $D(m)$ be the set of men matched to the women deleted from m's preference list. As $i \leq p - \Delta - 1$ at time t,

$$m' \in D(m) \Rightarrow m' \in W_{*[1,p-\Delta-1]}\,. \tag{2}$$

Let $R(m)$ be the set of men that are ranked better than m at least once by some woman. Thus, we have $D(m) \subseteq R(m)$. Say that m makes x proposals in total. Then, m was rejected or disengaged $x-1$ times due to a man from $R(m)$. As a woman only accepts better proposals in future, there are at least $|D(m)| + (x - 1)$ women that are matched to men in $R(m)$ when algorithm terminates. Thus,

$$\begin{aligned}
|D(m)| + (x-1) &\leq |R(m)| \\
(p - \Delta - 1) + (x-1) &\leq |R(m)| && ((2)) \\
p - \Delta + x - 2 &\leq |W_{*[1,p+\Delta]} \backslash m| && ((1)) \\
p - \Delta + x - 2 &\leq |W_{*[1,p+\Delta]}| - 1 && (\mathrm{c(m)} = \mathrm{p}) \\
p - \Delta + x - 2 &\leq p + 2\Delta - 1 && (\text{Lemma 6}) \\
x &\leq 3\Delta + 1\,.
\end{aligned}$$

◀

Let us now use our analysis to build a distributed version of the algorithm.

7 Distributed Algorithm

Before describing the distributed algorithm, we will compute some quantities that will give us some useful abstractions. Let us first compute Δ at every node in $\mathbb{M}$. We let $r(w_i, m)$ to be the rank assigned by w_i to $m \in \mathbb{M}$, i.e., $W_{ip} = m$ for $p = r(w_i, m)$.

▶ **Lemma 8.** *All nodes $m_j \in \mathbb{M}$ know $r(w_i, m_j)$ for all $i \in [1, n]$ after the following procedure is repeated for $p \in [0, n-1]$: for $k \in [1, n]$, w_k sends $r(w_k, m_{(k+p) \bmod n+1})$ to $m_{(k+p) \bmod n+1}$ in round p.*

Proof. Consider the nodes w_a and w_b where $a \neq b$. In any given round p the expression $(a+p) \bmod n+1 \neq (b+p) \bmod n+1$. Thus, messages sent by w_a and w_b in round p are sent to different nodes and received by them consequently. Also, the expression $(a+p) \bmod n+1$ covers every value in the range $[1, n]$ for a given a when p covers the range $[0, n-1]$. Thus, all nodes $w_a \in \mathbb{W}$ send $r(w_a, m_k)$ for $k \in [1, n]$. ◀

Now, using the previous lemma each node $m_j \in \mathbb{M}$ can compute the value of Δ.

▶ **Lemma 9.** *In $O(n)$ rounds, each node $m_j \in \mathbb{M}$ can compute the value of Δ.*

Proof. Using Lemma 8, any node m_j can compute the set $R_j = \{r(w_i, m_j) : i \in [1, n]\}$. Thus, m_j can compute $R_j^{max} = \max R_j$, $R_j^{min} = \min R_j$ and $\Delta_j = R_j^{max} - R_j^{min}$. Now in n rounds, each node m_j can send Δ_j to w_1. Then, w_1 can compute $\Delta = \max\{\Delta_j : j \in [1, n]\}$ and send it to all nodes m_j in another n rounds. ◀

Let us also compute the function c at the nodes $\mathbb{M}$, i.e., each node $m \in \mathbb{M}$ computes the set $\{c(m_j) : j \in [1, n]\}$. This function can be used to compute functions such as l locally that would be useful for distributed algorithm design.

▶ **Lemma 10.** *In $O(n)$ rounds, each node $m_j \in \mathbb{M}$ can compute the function c.*

Proof. Using Lemma 8, each node $m_j \in \mathbb{M}$ can compute the set $R_j = \{r(w_i, m_j) : i \in [1, n]\}$. Thus, m_j can compute $c(m_j) = \min R_j$.

We define a circular path P of nodes from $\mathbb{M}$ so that the next node of a node m_j is $m_{j \bmod n+1}$. Each node m_j already knows $v_j = \langle m_j, c(m_j)\rangle$. Now, in 2 rounds each node $m_j \in P$ can send v_j to next node on P by forwarding it through $w_{j \bmod n+1}$. If this is done $n-1$ times, each value v_j reaches all the other $n-1$ nodes on path P and each node in $\mathbb{M}$ stores the complete function c. ◀

▶ **Corollary 11.** *In $O(n)$ rounds, each node $m_j \in \mathbb{M}$ can compute the function l.*

Proof. Using Lemma 10, each node $m_j \in \mathbb{M}$ computes the function c in $O(n)$ rounds. Now, l can computed locally by each node using the function c and a deterministic tie-breaking for nodes m_j that have same value of $c(m_j)$. ◀

Once the nodes $\mathbb{M}$ know the functions c and l, they can also compute the following functions locally.

1. We define the sequence L as the nodes $\mathbb{M}$ arranged in ascending order of values assigned by function l. The value of $Next(i)$ is j so that the node m_j is next to the node m_i in L. Similarly, the value of $Prev(i)$ is j so that the node m_j occurs just before m_i in L. We use $\perp$ to point past end or before start of L. Formally,

$$Next(i) = \begin{cases} l^{-1}(l(i)+1) & \text{if } l(i) < n-1, \\ \perp & \text{otherwise} \end{cases}$$

and

$$Prev(i) = \begin{cases} l^{-1}(l(i)-1) & \text{if } l(i) > 1, \\ \perp & \text{otherwise}. \end{cases}$$

2. Consider the sequence L. We call a node m_j a *pivot* if m_j is first node in L or $c(m_j) > c(m_{Prev(j)})$. If a node m_j is a pivot, then $F(j)$ points to itself. Otherwise, $F(j)$ points to the next pivot node in L ($\perp$ if there is no next pivot in L). Formally,

$$F(i) = \begin{cases} i & \text{if } i = l^{-1}(\min\{l(j) : c(m_j) = c(m_i)\}), \\ l^{-1}(\min\{l(j) : c(m_j) > c(m_i)\}) & \text{if } \{l(j) : c(m_j) > c(m_i)\} \neq \emptyset, \\ \perp & \text{otherwise}. \end{cases}$$

3. If m_i is a pivot node, the value of $Wait(i)$ is the number of men m_j that have $c(m_j) = c(m_i)$. Otherwise, it is zero.

$$Wait(i) = \begin{cases} |\{m_j : c(m_j) = c(m_i)\}| & \text{if } i = l^{-1}(\min\{l(j) : c(m_j) = c(m_i)\}), \\ 0 & \text{otherwise} \end{cases}$$

Our goal is to design a distributed algorithm that simulates the sequential algorithm in constant number of rounds per proposal. The broad idea of the algorithm can be explained as follows. The algorithm matches men on the left side of sequence L and simultaneously deletes elements from the preference lists of men on the right side of sequence L. As the algorithm proceeds, the left part of sequence L consisting of matched men grows. The function F is used to pass on the deletion information from the first part to the second part. The function *Wait* is used to ensure that before a proposal is made, deletion of elements from the preference lists of men until the next pivot is finished. The functions $Next$ and $Prev$ are used to put the nodes into sequence L initially.

The precise description is in Algorithm 2 for a node m_i and in Algorithm 3 for a node w_i. The variable *counter* increments by 1 in each round. The list of unmatched men as per their order in the sequence L is maintained by the variables *next* and *prev*. The variable *match* stores the current partner ($\perp$ if none). The algorithm proceeds in *phases*. Each phase lasts eight rounds. A proposal is only sent in the first six rounds of the phase (*counter* mod $8 < 6$) where as the deletion of women from the preference lists occurs in the last two rounds of the phase (*counter* mod $8 \geq 6$). If a free woman receives a proposal, the initiation of her deletion from the preference lists also occurs during the first six rounds. The variable D contains the woman that should be checked for deletion in the next round and is updated during the last two rounds of the phase.

Let us now check if the distributed algorithm achieves its goal of simulating the sequential algorithm in a constant number of rounds per proposal.

8 Analysis of the Distributed Algorithm

Let us establish some helpful notations first. X_i^t is the value of the variable X stored by node m_i at the start of round $t \geq 0$ (loop at Line 4 when *counter* $= t$). $\widehat{X}_i^t$ is the value of the variable X stored by node w_i at the start of round t. The notation $p(t)$ represents the number of $\langle propose \rangle$ messages sent until round t starts. $DM(t)$ is the matching stored by the variables $match_i^t$ for $i \in [1, n]$ at the start of round t. Similarly, $\widehat{DM}(t)$ is the matching stored by the variables $\widehat{match}_i^t$ for $i \in [1, n]$ at the start of round t. $SM(q)$ is the matching

Algorithm 2: The algorithm executed by a node m_i to compute its match.

```
1  next ← Next(i), prev ← Prev(i);
2  wait ← Wait(i), match ← ⊥;
3  D ← ⊥, r ← ⊥, accepted ← false;
4  for counter ← 0 to 8(3Δ + 2)n − 1 do
5      if counter mod 8 = 0 then
6          if match = ⊥ and prev = ⊥ then
7              if wait > 0 then
8                  wait ← wait − 1;
9              else
10                 Send ⟨propose⟩ to next woman w_j available in prefrence list;
11     else if counter mod 8 = 1 then
12         if Received ⟨accept, x⟩ from w_j then
13             r ← x;
14             match ← j;
15             accepted ← true;
16     else if counter mod 8 = 2 and accepted = true then
17         Send ⟨SetPrev, r⟩ to w_next if next ≠ ⊥;
18     else if counter mod 8 = 3 then
19         if Received ⟨SetPrev, r⟩ then
20             prev ← r;
21     else if counter mod 8 = 4 and accepted = true then
22         if r = ⊥ and next ≠ ⊥ and F(next) ≠ ⊥ then
23             Send ⟨SetD, (w_match, c(m_Prev(next)) + Δ)⟩ to w_F(next);
24         else
25             Send ⟨SetNext, next⟩ to w_r if r ≠ ⊥;
26         next ← ⊥;
27         accepted ← false;
28     else if counter mod 8 = 5 then
29         if Received ⟨SetD, x⟩ then
30             D ← x;
31         if Received ⟨SetNext, x⟩ then
32             next ← x;
33             match ← ⊥;
34     else if counter mod 8 = 6 and D ≠ ⊥ then
35         Say D = (w_j, x);
36         Remove w_j from preference list if c(m_i) > x;
37         Send ⟨SetD, D⟩ to w_Next(i) if Next(i) ≠ ⊥;
38         D ← ⊥;
39     else // counter mod 8 = 7
40         if Received ⟨SetD, x⟩ then
41             D ← x;
```

Algorithm 3: The algorithm executed by a node w_i to compute its match.

```
match ← ⊥;
if Received ⟨propose⟩ from m_j then
    if match = ⊥ or m_j ranked higher than m_match then
        Send ⟨accept, match⟩ to m_j in the next round;
        match ← j;
if Received a message X other than ⟨propose⟩ then
    Forward X to m_i in the next round;
```

computed by Algorithm 1 after q proposals are sent. Given a tuple $D = (a, b)$, $D.1 = a$ and $D.2 = b$. The notation $i(w)$ represents the value of i in Algorithm 1 when $w \in \mathbb{W}$ receives its first proposal. Consider the sequence of values $wait_i^t$ for a given t and $i \in [1, n]$ in the same order as L. $fw(t)$ is j so that $wait_j^t$ is the first positive element of the sequence ($\perp$ if all $wait_i^t \leq 0$). We define the following invariants $Inv(t)$ where $t \geq 0$ is the first round of a phase, i.e., $t = 8k$ for $k \in [0, (3\Delta + 2)n - 1]$. Note that the algorithm terminates after $8(3\Delta + 2)$ rounds. For convenience, however, we will use the start of round $e = 8(3\Delta + 2)$ to refer the invariants just after the last round.

1. At most one $\langle propose \rangle$ message is sent in a phase. If the q^{th} proposal in the sequential algorithm is sent by m to w, then the q^{th} $\langle propose \rangle$ message is sent by m to w for $q \leq p(t)$. Moreover, $DM(t) = \widehat{DM}(t) = SM(p(t))$.
2. If m_i is matched or $match_i^t \neq \perp$, then $prev_i^t = next_i^t = \perp$. Otherwise, the value $next_i^t$ points to the first unmatched node after m_i in the sequence L ($\perp$ if it does not exists). Similarly, the value $prev_i^t$ points to the last unmatched node before m_i in the sequence L ($\perp$ if it does not exists). Formally, we have the following for $match_i^t = \perp$.

$$next_i^t = \begin{cases} l^{-1}(\min\{l(j) : l(j) > l(i), match_j^t = \perp\}) & \text{if } \{l(j) : l(j) > l(i), match_j^t = \perp\} \neq \emptyset, \\ \perp & \text{otherwise} \end{cases}$$

$$prev_i^t = \begin{cases} l^{-1}(\max\{l(j) : l(j) < l(i), match_j^t = \perp\}) & \text{if } \{l(j) : l(j) < l(i), match_j^t = \perp\} \neq \emptyset, \\ \perp & \text{otherwise.} \end{cases}$$

3. If $\widehat{match}_i^t \neq \perp$, then $D_j^t.1 = w_i$ for at most one j. If such j exists, then $D_j^t.2 = i(w_i) + \Delta$. Also, m_k has deleted w_i from its preference list if $c(m_k) > i(w_i) + \Delta$ and $l(k) < l(j)$. If there is no j so that $D_j^t.1 = w_i$, then m_k has deleted w_i from its preference list if $c(m_k) > i(w_i) + \Delta$.
4. $D_i^t = \perp$ for the following nodes m_i.

$$m_i \in \begin{cases} \{m_i : l(i) \leq l(fw(t)) + Wait(fw(t)) - wait_{fw(t)}^t\} & \text{if } fw(t) \neq \perp, \\ \mathbb{M} & \text{if } fw(t) = \perp \end{cases}$$

5. Let m_i be the node so that $match_i^t = \perp$ and $prev_i^t = \perp$. If m_i sends the message $\langle propose \rangle$ in round t and $next_i^t \neq \perp$, then the nodes m_j with $l(j) \geq l(next_i^t)$ have not sent a $\langle propose \rangle$ message until round t and $wait_j^t = Wait(j)$.

For brevity, we will refer to the individual invariants as $Inv(t).1$, $Inv(t).2$ and so on. We can easily show the following assuming that the above invariants are true.

▶ **Lemma 12.** *If $Inv(t)$ holds for all $t = 8k$ where $k \geq 0$, then there is a $k \leq (3\Delta + 2)$ for which $match_i^t \neq \perp$ for all $i \in [1, n]$ and Algorithm 2, Algorithm 3 terminate in a stable matching.*

Proof. Consider the start of phase in round r when $match_j^r = \perp$ for some j. Using $Inv(r).2$, there is a unique i for which $match_i^r = \perp$ and $prev_i^r = \perp$. Now, either $wait_i^{r+1} = wait_i^r - 1$ or m_i sends a $\langle propose \rangle$ message. As $\Sigma_{i=1}^n Wait(i) = n$, there are at most n phases where $wait_i^r > 0$. This leaves at least $(3\Delta + 2)n - n = (3\Delta + 1)n$ phases in which $\langle propose \rangle$ can be sent. Using $Inv(r).1$ and Lemma 7, these are sufficient until everyone is matched in a round $y = 8k$ for $k \leq (3\Delta + 2)$.

Note that the sequential algorithm terminates as soon as all the men are matched. Using Lemma 5, $SM(p(y)) = DM(y) = \widehat{DM}(y)$ is a stable matching. The matching remains the same until termination as no further $\langle propose \rangle$ messages are sent. ◀

Now, we only need to show that $Inv(t)$ holds. We first give the following helper lemma that gives the changes during the last two rounds of the phase.

▶ **Lemma 13.** *Let $t = 8k$ for $k \in [0, (3\Delta+2)n-1]$ be the first round of a phase. If $D_j^{t+6} \neq \perp$, $Prev(j) \neq \perp$, then $D_j^{t+8} = D_{Prev(j)}^{t+6}$.*

Proof. If $D_j^{t+6} \neq \perp$, $D_k^{t+6} \neq \perp$, $Next(j) \neq \perp$ and $Next(k) \neq \perp$ for $j \neq k$, then nodes m_j, m_k send $\langle SetD, D_j^{t+6} \rangle$ and $\langle SetD, D_k^{t+6} \rangle$ respectively to $w_{Next(j)}$ and $w_{Next(k)}$ in round $t + 6$. If $j \neq k$, then $Next(j) \neq Next(k)$ by definition of function $Next$. Thus, these messages are received by the nodes $w_{Next(j)}$ and $w_{Next(k)}$ which forward the message to $m_{Next(j)}$ and $m_{Next(k)}$ in round $t+7$. Upon receiving the message, the nodes $m_{Next(j)}$ and $m_{Next(k)}$ simply set their D to D_j^{t+6}, D_k^{t+6} respectively as received in the message. Thus, if m_j receives a message in round $t + 7$, then $D_j^{t+8} = D_{Prev(j)}^{t+6}$. If m_j does not receive a message in round $t + 7$, then $D_{Prev(j)}^{t+6} = \perp$ and $D_j^{t+8} = D_j^{t+7} = \perp$ as well. ◀

We will first show the base case of induction and later the induction step.

▶ **Lemma 14.** *$Inv(0)$ holds.*

Proof.
$Inv(0).1$. Both $DM(0)$ and $\widehat{DM}(0)$ are empty matchings by initialization which is same as $\overline{SM(p(0))}$.

$Inv(0).2$. The initialization values $Next(i)$ and $Prev(i)$ satisfy the invariant for $i \in [1, n]$.

$Inv(0).3$. By initialization, $\widehat{match_i}^0 = \perp$ for all $i \in [1, n]$. As nodes start with complete preference lists (without any deleted entries), the invariant holds.

$Inv(0).4$. By definition of $Wait$ we have $wait_a^0 = Wait(a) > 0$ where $a = l^{-1}(1)$. As $D_a^0 = \perp$ by initialization, the invariant follows.

$Inv(0).5$. By initialization, $wait_a^0 = Wait(a) > 0$ where $a = l^{-1}(1)$ and propose is not sent in the first round. Thus, invariant holds as precondition is not true. ◀

▶ **Lemma 15.** *If $Inv(t)$ holds for $t = 8k$, $k \in [0, (3\Delta + 2)n - 1]$ and there exists $i \in [1, n]$ so that $match_i^t = \perp$, then $Inv(t + 8)$ holds.*

Proof. Using $Inv(t).2$, there is exactly one man m_i so that $match_i^t = \perp$ and $prev_i^t = \perp$. To show $Inv(t + 8)$ we consider two cases: $wait_i^t \not> 0$ and $wait_i^t > 0$.

$Inv(t + 8).1$, $Inv(t + 8).2$ when $wait_i^t \not> 0$. As $wait_i^t \not> 0$, m_i sends a $\langle propose \rangle$ message to a woman w_q. Using $Inv(t).1$, the set of unmatched men are the unmatched men in the

matching $SM(p(t))$. Say that the $(p(t)+1)^{th}$ proposal in the sequential algorithm is sent by m to w. Using $Inv(t).2$, $m_i = m$ and all the men m_j with $l(j) < l(i)$ are matched. Thus, the men m_j, $l(j) < l(i)$ have sent their first $\langle propose \rangle$ message in round q (say) and $wait_j^q \not> 0$. As the variable *wait* is never incremented, $wait_j^t \not> 0$ for $l(j) < l(i)$. Using $Inv(t).4$ and the assumption $wait_i^t \not> 0$, $D_j^t = \perp$ for $l(j) \leq l(i)$. Then, we can use $Inv(t).3$ to conclude that a woman w' is deleted from m_i's preference list if w' is matched and $c(m_i) > i(w') + \Delta$. Note that a woman w' is also deleted from m's preference list when m sends a proposal in the sequential algorithm if w' is matched and $c(m) > i(w') + \Delta$. Thus, we have $w = w_q$.

As no other messages are sent in round t, $\langle propose \rangle$ message is received by w_q. As in the sequential algorithm, w_q accepts the proposal if it is better. If w_q does not reply to $\langle propose \rangle$ message, then no change to variables *match*, *next* and *prev* are done and both $Inv(t+8).1$, $Inv(t+8).2$ hold. If w_q replies with $\langle accept, x \rangle$ message, then we have the following cases depending on the values of x and $next_i^t$.

Case (a): $x = \perp$, $next_i^t = \perp$. The changes to the variables *match*, *next* and *prev* until the end of the phase are as follows.

1. $\widehat{match}_q^{t+2} = i$ as w_q updates its *match* variable in round $t+1$.
2. $match_i^{t+2} = q$ as m_i receives $\langle accept, x \rangle$ in round $t+1$.
3. $next_i^{t+5} = \perp$ as m_i sets the variable in round $t+4$.

Thus,

1. $\widehat{match}_q^{t+8} = i$,
2. $match_i^{t+8} = q \neq \perp$,
3. $next_i^{t+8} = \perp$, and
4. $prev_i^{t+8} = \perp$ as $prev_i^t = \perp$ by assumption.

So $Inv(t+8).1$, $Inv(t+8).2$ hold in this case.

Case (b): $x = \perp$, $next_i^t \neq \perp$. The changes to the variables *match*, *next* and *prev* until the end of the phase are as follows.

1. $\widehat{match}_q^{t+2} = i$ as w_q updates its *match* variable in round $t+1$.
2. $match_i^{t+2} = q$ as m_i receives $\langle accept, x \rangle$ in round $t+1$.
3. $prev_a^{t+4} = \perp$ where $a = next_i^t$ as m_i sends $\langle SetPrev, \perp \rangle$ to w_a in round $t+2$ which forwards it to m_a in round $t+3$.
4. $next_i^{t+5} = \perp$ as m_i sets the variable in round $t+4$.

Thus,

1. $\widehat{match}_q^{t+8} = i$,
2. $match_i^{t+8} = q \neq \perp$,
3. $next_i^{t+8} = \perp$,
4. $prev_i^{t+8} = \perp$ as $prev_i^t = \perp$ by assumption, and
5. $prev_a^{t+8} = \perp$ where $a = next_i^t$.

So $Inv(t+8).1$, $Inv(t+8).2$ hold in this case.

Case (c): $x \neq \perp$, $next_i^t = \perp$. The changes to the variables *match*, *next* and *prev* until the end of the phase are as follows.

1. $\widehat{match}_q^{t+2} = i$ as w_q updates its *match* variable in round $t+1$.
2. $match_i^{t+2} = q$ as m_i receives $\langle accept, x \rangle$ in round $t+1$.
3. $next_i^{t+5} = \perp$ as m_i sets the variable in round $t+4$.
4. $next_r^{t+6} = \perp$ and $match_r^{t+6} = \perp$, where $r = x$ as m_i sends $\langle SetNext, \perp \rangle$ to w_r in round $t+4$ which forwards it to m_r in round $t+5$.

Thus, we have the following for $r = x$.

1. $\widehat{match}_q^{t+8} = i$,

2. $match_i^{t+8} = q \neq \perp$, $next_i^{t+8} = \perp$, $prev_i^{t+8} = \perp$ as before,
3. $next_r^{t+8} = \perp$, $match_r^{t+8} = \perp$, and
4. $prev_r^{t+8} = \perp$ as $match_r^t \neq \perp$ by assumption, so $prev_r^t = \perp$ by $Inv(t).2$.

As before, $Inv(t+8).1$ holds in this case as well. As $next_i^t = \perp$, m_r is the only unmatched man left and $Inv(t+8).2$ also holds.

Case (d): $x \neq \perp$, $next_i^t \neq \perp$. The changes to the variables *match*, *next* and *prev* until the end of the phase are as follows.

1. $\widehat{match}_q^{t+2} = i$ as w_q updates its *match* variable in round $t+1$.
2. $match_i^{t+2} = q$ as m_i receives $\langle accept, x\rangle$ in round $t+1$.
3. $prev_a^{t+4} = r$ where $a = next_i^t$ and $r = x$ as m_i sends $\langle SetPrev, r\rangle$ to w_a in round $t+2$ which forwards it to m_a in round $t+3$.
4. $next_i^{t+5} = \perp$ as m_i sets the variable in round $t+4$.
5. $next_r^{t+6} = a$ and $match_r^{t+6} = \perp$, where $a = next_i^t$ and $r = x$ as m_i sends $\langle SetNext, a\rangle$ to w_r in round $t+4$ which forwards it to m_r in round $t+5$.

Thus, we have the following for $r = x$ and $a = next_i^t$.

1. $\widehat{match}_q^{t+8} = i$,
2. $match_i^{t+8} = q \neq \perp$, $next_i^{t+8} = \perp$, $prev_i^{t+8} = \perp$ as before,
3. $prev_r^{t+8} = \perp$, $match_r^{t+8} = \perp$ as before, and
4. $next_r^{t+8} = a$.

Using $Inv(t).5$, $l(r) < l(a)$ and these changes are sufficient for $Inv(t+8).2$ to hold. Also, $Inv(t+8).1$ holds as well.

$Inv(t+8).3$ when $wait_i^t \not> 0$. The invariant needs to be checked only when the value of variable D changes. This only happens upon receiving the message $\langle SetD, x\rangle$.

If the node m_i receives $\langle accept, y\rangle$ in round $t+1$ with $y \neq \perp$, then $\langle SetD, x\rangle$ is only sent in round $t+6$ by a node m_j if $D_j^{t+6} \neq \perp$. Using Lemma 13, if $\widehat{match}_a^t \neq \perp$, $D_l^t.1 = w_a$ and $Next(l) \neq \perp$, then $D_{Next(l)}^{t+8}.1 = w_a$. Using $Inv(t).3$, we only need to check deletion of w_a from m_l's list. Before m_l sends $\langle SetD, x\rangle$, it deletes $D_l^t.1 = w_a$ if $c(m_l) > D_l^t.2$. Using $Inv(t).3$, we have $D_l^t.2 = i(w_a) + \Delta$ and $Inv(t+8).3$ holds.

If the node m_i receives $\langle accept, \perp\rangle$ in round $t+1$ from w_q, then $\langle SetD, x\rangle$ is only sent in rounds $t+4$ and $t+6$. In round $t+4$, m_i sends $\langle SetD, x\rangle$ to $w_{F(a)}$ where $a = next_i^{t+4}$ and $x = (w_q, c(m_{Prev(a)}) + \Delta)$. Using $Inv(t).5$, all nodes m_j with $l(j) \geq l(a)$ never sent a $\langle propose\rangle$ message before and all the other nodes m_k already sent a $\langle propose\rangle$ message as they are matched. Using $Inv(t).1$ and $Inv(t+8).1$, $i(w_q) = \max\{c(m_j) : m_j \text{ sent } \langle propose\rangle \text{ until round } (t+4)\} = c(m_{Prev(a)})$. If m_a is a pivot node, then $F(a) = a$ by definition and $c(m_{F(a)}) = \min\{c(m_j) : c(m_j) > i(w_q)\} \leq \min\{c(m_j) : c(m_j) + \Delta > i(w_q)\}$. If m_a is not a pivot node, then $c(m_a) = c(m_{Prev(a)}) = i(w_q)$ and again $c(m_{F(a)}) = \min\{c(m_j) : c(m_j) > i(w_q)\} \leq \min\{c(m_j) : c(m_j) + \Delta > i(w_q)\}$. Therefore, $l(p) \geq l(F(a))$ if $c(m_p) > i(w_q) + \Delta$. Now we use Lemma 13 and $Inv(t).3$ as before to conclude that $Inv(t+8).3$ holds after $\langle SetD, x\rangle$ is sent in round $t+6$.

$Inv(t+8).4$ when $wait_i^t \not> 0$. If m_a for $a = next_i^t \neq \perp$ is a pivot node, then $F(a) = a$ and $Wait(a) > 0$ using definition of *Wait*. Using $Inv(t).5$, we conclude $wait_a^t = Wait(a) > 0$. As all nodes m_j with $l(j) < l(a)$ sent a $\langle propose\rangle$ message at least once, $wait_j^t \not> 0$. Thus, $a = F(a) = fw(t)$ if m_a is a pivot node. If m_a for $a = next_i^t \neq \perp$ is not a pivot node, then $Wait(k) = 0$ for $l(a) \leq l(k) < l(F(a))$ using definition of *Wait*. As before $wait_j^t \not> 0$ for all nodes m_j with $l(j) < l(a)$. Using $Inv(t).5$, we conclude that $wait_{F(a)}^t = Wait(F(a)) > 0$. Thus, we conclude that $F(a) = fw(t)$ irrespective of whether m_a is a pivot node or not.

Using $Inv(t).4$, we conclude $D_j^t = \perp$ for $l(j) \leq l(a)$. As *wait* is not changed in this phase, $Inv(t+8).4$ holds if $D_j^{t+8} = \perp$ for $l(j) \leq l(a)$. In round $t+4$, m_i may send $\langle SetD, x\rangle$ to w_a which then forwards it to m_a in round $t+5$ so $D_a^{t+6} \neq \perp$. Using Lemma 13, we again have $D_j^{t+8} = \perp$ for $l(j) \leq l(a)$.

If $next_i^t = \perp$, then all nodes m_k, $k \in [1, n]$ already sent a $\langle propose\rangle$ message at least once. Thus, $wait_k^t \not> 0$. Using $Inv(t).4$, $D_k^t = \perp$. As no $\langle SetD, x\rangle$ message is sent until round $t+8$ starts, $D_k^{t+8} = \perp$ as well.

$Inv(t+8).5$ when $wait_i^t \not> 0$. Using $Inv(t).5$, the nodes m_j with $l(j) \geq l(a)$ have not sent a $\langle propose\rangle$ message until round t for $a = next_i^t \neq \perp$. Thus, if a node m_k is matched, then $l(k) < l(a)$.

Now, if m_i receives $\langle accept, x\rangle$ in round $t+1$ with $x \neq \perp$, then $match_r^{t+8} = \perp$ for some r so that $l(r) < l(a)$. Using $Inv(t+8).2$, only m_r is the node that satisfies $match_r^{t+8} = \perp$ and $prev_r^{t+8} = \perp$. Using $Inv(t+8).2$, we also conclude that $next_r^{t+8} = a$. The nodes m_j with $l(j) \geq l(a)$ do not send a $\langle propose\rangle$ message in this phase nor change their *wait* variables. Using $Inv(t).5$, the nodes m_j with $l(j) \geq l(a)$ satisfy $wait_j^t = Wait(j)$. Thus, $Inv(t+8).5$ holds if a node m_r is rejected in this phase.

If m_i receives $\langle accept, x\rangle$ in round $t+1$ with $x = \perp$, then using $Inv(t+8).2$ we conclude that m_a is the only node that satisfies $match_a^{t+8} = \perp$ and $prev_a^{t+8} = \perp$. The nodes m_j with $l(j) \geq l(a)$ do not change their *match* variables in this phase, do not send a $\langle propose\rangle$ message in this phase and do not change their *wait* variables in this phase. Using $Inv(t).5$, the nodes m_j with $l(j) \geq l(a)$ satisfy $wait_j^{t+8} = Wait(j)$. In round $t+8$, only m_a may send a $\langle propose\rangle$ message Thus, if $next_a^{t+8} \neq \perp$, then the nodes m_j with $l(j) \geq l(next_a^{t+8})$ have not sent a $\langle propose\rangle$ message until round $t+8$ and have $wait_j^{t+8} = Wait(j)$.

If m_i does not receive a message from w_q in round $t+1$, then none of the nodes change their *match*, *prev*, *next* or *wait* variables. Also, only m_i may send a $\langle propose\rangle$ message in round $t+8$. Using $Inv(t).5$, the invariant also follows when m_i does not receive a message from w_q in round $t+1$.

$Inv(t+8)$ when $wait_i^t > 0$. In this case, m_i decrements its *wait* variable instead of sending a $\langle propose\rangle$ message. As a result, for all nodes m_j we have $next_j^{t+8} = next_j^t$, $prev_j^{t+8} = prev_j^t$ and $match_j^{t+8} = match_j^t$. Thus, there is nothing to check for $Inv(t+8).1$ and $Inv(t+8).2$. Using Lemma 13, $Inv(t+8).4$ holds as well because $wait_i^{t+8} = wait_i^t - 1$. Verifying $Inv(t+8).3$ in this case is same as verifying it when the $\langle propose\rangle$ message sent by m_i is replied with an $\langle accept, x\rangle$ where $x \neq \perp$.

We consider two cases to check for $Inv(t+8).5$. First: if $wait_i^t > 1$, then $wait_i^{t+8} > 0$. Thus, the message $\langle propose\rangle$ will not be sent in round $t+8$ and nothing needs to be checked. Second: if $wait_i^t = 1$, then $wait_i^{t+8} = 0$. Thus, the node m_i never sent a $\langle propose\rangle$ message until round t as otherwise $wait_i^t \not> 0$. Also, no node m_j with $l(j) \geq l(i)$ sent a $\langle propose\rangle$ message as otherwise m_i must have sent $\langle propose\rangle$ message at least once. Thus, no m_j with $l(j) > l(i)$ ever got matched and $prev_j^{t'} \neq \perp$ for the first round t' of any phase where $t' \leq t$. Thus, $wait_j^{t+8} = Wait(j)$ for $l(j) > l(i)$. As m_i sends $\langle propose\rangle$ in round $t+8$, we conclude that $Inv(t+8).5$ holds. ◀

▶ **Theorem 16.** *Algortihm 2 and Algorithm 3 compute a stable matching where each node needs $O(n(\Delta+1))$ messages of size $O(\log n)$.*

Proof. Using Lemma 14, Lemma 15 and Lemma 12, each node computes its partner in a stable matching in $O(n(\Delta+1))$ rounds. As a node receives at most one message of size $O(\log n)$ per round, the statement follows. ◀

9 Conclusion and Open Problems

We considered the distributed version of stable matching where preference lists of one set of participants are almost similar. Given the non-negative similarity parameter $\Delta \leq n-1$, any node can compute its partner in a stable matching by receiving $O(n(\Delta+1))$ values. Also, there is always a node that must receive $\Omega(n/\log n)$ values.

It still remains to find out if the above algorithm is optimal (up to a logarithmic factor) if Δ is not constant. Furthermore, it may also be interesting to have an algorithm that uses $O(1)$ sized messages instead of $O(\log n)$ sized messages.

References

1. Nir Amira, Ran Giladi, and Zvi Lotker. Distributed Weighted Stable Marriage Problem. In *17th International Colloquium on Structural Information and Communication Complexity (SIROCCO), Sirince, Turkey*, June 2010.
2. Ismel Brito and Pedro Meseguer. Distributed Stable Matching Problems. In *11th International Conference on Principles and Practice of Constraint Programming (CP), Sitges, Spain*, October 2005.
3. Ismel Brito and Pedro Meseguer. Distributed stable matching problems with ties and incomplete lists. In *12th International Conference on Principles and Practice of Constraint Programming (CP), Nantes, France*, September 2006.
4. Patrik Floréen, Petteri Kaski, Valentin Polishchuk, and Jukka Suomela. Almost Stable Matchings by Truncating the Gale–Shapley Algorithm. *Algorithmica*, September 2010.
5. D. Gale and L. S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, January 1962.
6. Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. The MIT Press, 1989.
7. Robert W. Irving, David F. Manlove, and Sandy Scott. The Stable Marriage Problem with Master Preference Lists. *Discrete Applied Mathematics*, August 2008.
8. Deepak Kapur and Mukkai S. Krishnamoorthy. Worst-case Choice for the Stable Marriage Problem. *Information Processing Letters*, July 1985.
9. Alex Kipnis and Boaz Patt-Shamir. A Note on Distributed Stable Matching. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS), Montreal, QC, Canada*, June 2009.
10. Eyal Kushilevitz. Communication Complexity. In *Advances in Computers*, volume 44. Elsevier, 1997.
11. Rafail Ostrovsky and Will Rosenbaum. Fast Distributed Almost Stable Matchings. In *34th ACM Symposium on Principles of Distributed Computing (PODC), Donostia-San Sebastián, Spain*, July 2015.
12. Michael J. Quinn. A Note on Two Parallel Algorithms to Solve the Stable Marriage Problem. *BIT Numerical Mathematics*, September 1985.
13. S. S. Tseng and R. C. T. Lee. A Parallel Algorithm to Solve the Stable Marriage Problem. *BIT Numerical Mathematics*, September 1984.

Time and Space Optimal Counting in Population Protocols

James Aspnes*[1], Joffroy Beauquier[2], Janna Burman†[3], and Devan Sohier[4]

1 **Yale University, New Haven, CT, USA**
`james.aspnes@gmail.com`
2 **Université Paris Sud, LRI, Paris, France**
`joffroy.beauquier@lri.fr`
3 **Université Paris Sud, LRI, Paris, France**
`janna.burman@lri.fr`
4 **Université de Versailles, LI-PaRAD, Versailles, France**
`devan.sohier@uvsq.fr`

Abstract

This work concerns the general issue of combined optimality in terms of time and space complexity. In this context, we study the problem of (exact) *counting* resource-limited and passively mobile nodes in the model of *population protocols*, in which the space complexity is crucial. The counted nodes are memory-limited anonymous devices (called agents) communicating asynchronously in pairs (according to a *fairness* condition). Moreover, we assume that these agents are prone to failures so that they cannot be correctly initialized.

This study considers two classical fairness conditions, and for each we investigate the issue of time optimality of counting given the optimal space per agent. In the case of randomly interacting agents (*probabilistic fairness*), as usual, the convergence time is measured in terms of *parallel time* (or parallel interactions), which is defined as the number of pairwise interactions until convergence, divided by n (the number of agents). In case of *weak fairness*, where it is only required that every pair of agents interacts infinitely often, the convergence time is defined in terms of *non-null transitions*, i.e, the transitions that affect the states of the interacting agents.

First, assuming probabilistic fairness, we present a "non-guessing" time optimal protocol of $O(n \log n)$ expected time given an optimal space of only one bit, and we prove the time optimality of this protocol. Then, for weak fairness, we show that a space optimal (*semi-uniform*) solution cannot converge faster than in $\Omega(2^n)$ time (non-null transitions). This result, together with the time complexity analysis of an already known space optimal protocol, shows that it is also optimal in time (given the optimal space constrains).

1998 ACM Subject Classification C.2.4 Distributed Systems, I.1.2 Algorithms

Keywords and phrases networks of passively mobile agents/sensors, population protocols, counting, anonymous non-initialized agents, time and space complexity, lower bounds, probabilistic/weak fairness

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.13

* The work of this author is supported in part by NSF grants CCF-1650596 and CCF-1637385.
† Contact author.

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 13; pp. 13:1–13:17


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In this paper we are interested in the determination of the *exact* number of nodes in a mobile sensor network. In the considered networks, sensors are typically attached to mobile supports (vehicles, animals, people, etc.) moving in an unpredictable way. Moreover, nodes may be deployed at large scale. Therefore they should be cheap and consequently can be prone to failures. One can think of sensors attached to zebras (ZebraNet [22]), pigeons (Pigeon Air Patrol [28]), or public transport vehicles (EMMA project [24]). In this context, counting can be part of the task being realized (How many animals have a temperature exceeding some bound?), or part of the managing of the network (Should some nodes be added or replaced?). In relation with the domain of application we are looking at, we consider that the nodes are anonymous, undistinguishable, have a bounded memory and poor communication capabilities (no broadcast; only a pairwise communication when two nodes come close enough to each other). A distributed computing model that fits this description is the model of *population protocols* (PP) [4].

In PP, mobile nodes, which are called agents, can be represented as finite state transition systems. One can imagine that, when two agents are close enough, they interact and the effect of the interaction is a transition with a possible change of states. In this work we study the case of *symmetric protocols*, where two agents in a transition are indistinguishable if their states are identical (thus, their states have to be identical also after the transition). This assumption makes the protocol design more difficult than in the asymmetric case (as it restrains the set of possible transition rules), but provides a more general solution (correct in both cases).

The mobility and the resulting interactions of agents are completely asynchronous and modeled in a very general way - by a *fairness* assumption. Here, we study the problem of counting considering two classical fairness assumptions. One ensures that each pair of agents is drawn uniformly at random for each interaction, and the other, weaker assumption (called here *weak*), ensures only that every pair of agents interact infinitely often. While the probabilistic fairness captures the randomization inherent to many real systems, weak fairness only ensures progress of system entities (see Sec. 2 for an illustrating example).

As the agents are likely to be cheap and prone to failures (memory corruption, crash failures, etc.), re-counting may be required frequently. Since the population of agents may be very large, re-initialization may be infeasible. Hence, it is natural to assume that the agents are not initialized (i.e., an agent can be initially in any possible state). However, it is easy to prove that, in this case, counting in PP is impossible [11]. The solution is to use only one initialized and distinguishable agent, called the *base station* (BST). In this work, for the first time, we also prove, the necessity of such an agent being distinguishable (in case of symmetric protocols; see Sec. 4). BST is also the agent that eventually obtains the correct count of the population. Thus the considered protocols are *semi-uniform*, in the sense that all the agents, except BST, are a priori undistinguishable and execute the same protocol, for any population size n and upper bound P on n (see Sec. 2 for a formal definition).

In this context, previous works [11, 20, 9] and a companion paper [10] study the issue of space complexity (of the counted nodes), a factor that is particulary important in the considered large-scale and unreliable networks. For instance, [11] shows that under weak fairness, P (or more) agents cannot be counted with strictly less than P states per counted agent, by deterministic protocols (considered here as well). Here, as a by-product, we present an alternative proof of this result, for the case of symmetric protocols (see Proposition 10).

However, as shown in [10], under probabilistic or *global fairness*[1], counting can be performed with only two states (one bit) per counted agent. [10] presents two space optimal solutions to counting in PP, one under weak fairness and the other under global fairness (the latter is also correct under probabilistic fairness). The solution for global fairness uses a memory of only one bit per agent, while the solution for weak fairness needs $\log P$ bits (P states) per counted agent. In the latter case, as the number P of states for a given counting protocol is fixed, it represents an upper bound on the size of populations to which the protocol applies, and is considered as an explicit parameter.

In addition to the state space optimality, this paper raises the issue of the convergence time. Our objective is to determine the best guarantees on the convergence times given the established necessary minimum on the memory. To obtain this goal, we show, in particular, that the convergence times of the two previous space optimal solutions (in [10]) are exponential. In Sec. 3, for reasons explained there, we restrict our attention to so called "non-guessing" counting protocols. We prove that any such state optimal counting protocol, correct under probabilistic fairness, converges in $\Omega(n \log n)$ expected parallel interactions (Sec. 3.1). In Sec. 3.2, we propose a new state optimal protocol fitting this complexity.

In the case of weak fairness, in Sec. 4, we show that a space optimal solution requires an exponential convergence time (in terms of non-null transitions). In particular, this result shows that the space optimal protocol under weak fairness in [10] is time optimal among the space optimal semi-uniform protocols.

Related Work

We provide here the most relevant and recent works. Please, refer to [10] for additional related literature on the subject.

Considering PP, [11, 20, 9] proposed efficient counting protocols in terms of exact state space that were improved in [10] by space optimal solutions. For weak fairness, the protocol proposed in [20] uses $2P$ states per agent (only 1 bit more than the optimal) and converges in only $O(\log n)$ asynchronous rounds (a round being a shortest fragment of execution where each agent interacts with each other). This presents an interesting trade-off for counting in PP. A recent work [27] studies a problem related to counting in random PP, where agents should determine the difference between the number of agents started in state A and the number of those started in state B. In contrast with the current work, [27] assumes initialized agents, but similarly to the current work, it investigates the efficiency in terms of both time and space. It presents an $O(n^{3/2})$-state population protocol that allows each agent to converge to the exact solution by interacting no more than $O(\log n)$ times. Additional very recent works (as [1, 17, 2, 3]) jointly contribute to the time and space trade-offs study of fundamental tasks, as majority and leader election, in random PP. For example, [17] shows that it is impossible to achieve sub-linear leader election with only constant state space per agent, but due to [2] this problem can be solved in $O(\log^3 n)$ time with $O(\log^3 n)$ states. For majority, sub-linear time is impossible for protocols with at most four states per node, while there exists a poly-logarithmic time protocol which requires a linear in n state space [3]. [1] presents additional upper and lower bounds for these tasks that highlight a time complexity separation between $O(\log \log n)$ and $\Theta(\log^2 n)$ state space for both majority and

[1] Global fairness can be viewed as simulating probabilistic one without introducing randomization explicitly. One can see probabilistic fairness as a quantitative version of the global one. Moreover, it allows to analyze protocols' time complexity, what is impossible in general with global fairness (see Sec. 2.)

leader election. The present work contributes to the general study of time-space trade-offs in the case of counting.

In the context of dynamic networks with anonymous nodes, recent works [25, 13, 12, 26, 14], study the counting problem in the *synchronous* model of dynamic graphs. PP can be also represented by dynamic graphs, but is a completely asynchronous model. Moreover, in contrast with the current study, protocols in these works require that all nodes are initialized. These essential differences make the techniques (e.g., termination detection), extensively used there, inappropriate in our case. Note however that their protocols determining the exact count have exponential convergence time, except the results of [14] considering more restricted networks. In some sense this supports the results presented here for weak fairness. [14] presents interesting time complexity lower bounds for counting, but considers networks where anonymous nodes can communicate (by broadcast) *any* amount of data and diffuse it to all other nodes in a *constant* time w.r.t. n, what is of course impossible in our context.

Due to the difficulty of the problem, a lot of work has been devoted to design protocols counting *approximately* the number of network nodes (see, e.g., [19, 23, 18, 29, 8, 31, 6]). These protocols use gossiping and probabilistic methods, like probabilistic polling, random walks, epidemic-based approaches, and also exploit classical results on order statistics to infer an estimated number of the nodes. Here, we consider only deterministic protocols for exact counting.

2 Model and Notations

A system consists of a collection $\mathcal{A}$ of pairwise interacting agents, also called a population. Each agent represents a finite state sensing and communicating mobile device. Among the agents, there is a distinguishable agent called the *base station* (BST), which can be as powerful as needed, in contrast with the resource-limited non-BST agents. The non-BST agents are also called *mobile*, interchangeably. The size of the population is the number of mobile agents, denoted by n, and is unknown (a priori) to the agents.

A *(population) protocol* can be modeled as a finite transition system whose states are called *configurations*. A *configuration* is a vector of states of all the agents. Each agent has a state taken from a finite set of states, the same for all mobile agents (denoted Q), but generally different for BST.

In this transition system, every transition $C \to C'$ between two configurations C and C' is modeled by a single *transition* between two agents happening during an interaction. That is, when two agents x, in state p, and y, in state q, in C, interact (meet), they execute a transition rule $(p, q) \to (p', q')$. As a result, in C', x changes its state from p to p' and y from q to q'. If $p = p'$ and $q = q'$, the corresponding transition is called *null* (such transitions are specified by default), and non-null otherwise. For simplicity, in some cases, we do not present protocols under the form of transition rules, but under the equivalent form of a pseudo-code. If there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $C_i \to C_{i+1}$ for all $i, 0 \leq i < k$, we say that C' *is reachable* from C, denoted $C \xrightarrow{*} C'$.

The transition rules of a protocol are *deterministic*, if for every pair of states (p, q), there is exactly one (p', q') such that $(p, q) \to (p', q')$. We consider only deterministic transitions and thus, only *deterministic protocols*. Transitions and protocols can be *symmetric* or *asymmetric*. Symmetric means that, if $(p, q) \to (p', q')$ is a transition rule, then $(q, p) \to (q', p')$ is also a transition rule. In particular, if $(p, p) \to (p', q')$ is symmetric, $p' = q'$. Asymmetric is the contrary of symmetric.

Let $(p_1, q_1) \to (p_2, q_2)$, $(p_2, q_2) \to (p_3, q_3)$, …, $(p_{k-2}, q_{k-2}) \to (p_{k-1}, q_{k-1})$, $(p_{k-1}, q_{k-1}) \to$

(p_k, q_k) be the transition rules of a protocol. Then, we shortly write $(p_1, q_1) \xrightarrow{*} (p_k, q_k)$ to denote a possible sequence of these transition rules, which can be applied (in the same order) on two agents in states p_1 and q_1, making them interact repeatedly until their states change to p_k and q_k, respectively. We sometimes call an agent in state p a p-state agent, or just p agent.

An *execution* of a protocol is an infinite sequence of configurations $C_0, C_1, C_2, \ldots$ such that C_0 is the starting configuration and for each $i \geq 0$, $C_i \to C_{i+1}$. In a real distributed execution, interactions of distinct agents are independent and could take place simultaneously (in parallel), but when writing down an execution we can order those simultaneous interactions arbitrarily.

An execution is said *weakly fair*, if every pair of agents in $\mathcal{A}$ interacts infinitely often. An execution is said *probabilistically fair*, if, for each interaction in the execution, a pair of agents in $\mathcal{A}$ is chosen uniformly at random. An execution is said *globally fair*, if for every two configurations C and C' such that $C \to C'$, if C occurs infinitely often in the execution, then C' also occurs infinitely often in the execution. This also implies that, if in an execution there is an infinitely often reachable configuration, then it is infinitely often reached [5]. Global fairness can be viewed as simulating randomized systems without introducing randomization explicitly (any probabilistically fair execution is globally fair with probability 1 [21]).

A simple example allows to understand better the difference between weak and global (or probabilistic) fairness. Consider a population of 3 agents. Each agent can be white or black, and initially one agent is black and the two others are white. Consider also the protocol in which, when two white agents interact, they become both black and when two agents of different colors interact, they exchange their colors. It is easy to see that there is an infinite weakly fair execution in which there is always one black and two white agents (the black color "jump" indefinitely from agent to agent). At the contrary, every globally fair execution terminates in a configuration in which the 3 agents are black, because otherwise there would be infinitely many configurations during an execution from which the "all black" configuration could be reached, without ever being reached (contradicting global fairness).

A *problem* is defined by a predicate $\mathcal{D}$ on executions. A population protocol $\mathcal{P}$ is said to *solve a problem* $\mathcal{D}$, if and only if every execution of $\mathcal{P}$ satisfies the conditions defining $\mathcal{D}$. The problem of *counting* is defined by the following condition: eventually, in any execution, there is at least one agent (BST, in our case) obtaining a value of n in some (estimate) variable (called c in the following) and this value does not change. Note that the counting predicate has to be satisfied only eventually (and forever after). When it happens, we say that the protocol has *converged*. A protocol is called *silent*, if in any execution, eventually, no state of an agent changes [15].

In the case of probabilistic fairness, the *convergence time* of a protocol is measured in terms of *parallel time* or *parallel interactions*, i.e., the independent interactions (of distinct agents) occurring in parallel. It is customary to define one *unit of parallel time* as n consecutive interactions in a probabilistically fair execution. Then, in this case, the convergence time of a protocol is defined by the expected number of parallel time units in a fair execution till convergence. Moreover, under probabilistic fairness, it appears that technically (in the context of this specific work), we can perform the convergence time analysis in terms of transitions involving BST. It is easy to see that this corresponds to the (asymptotic) expected convergence time in terms of parallel time units.

In the case of weak fairness, the convergence time of a protocol is defined as the maximum number of non-null transitions in a fair execution till convergence.

We consider only *semi-uniform* protocols (cf. [16, 30]) in the sense that all agents, except BST (whence semi-), are a priori indistinguishable and interact according to the same

transition rules. Moreover, the protocol functions similarly for any n and any upper bound P on n. More formally, we can define a semi-uniform protocol so:

▶ **Definition 1.** A protocol $\mathcal{PP}$ is called *semi-uniform* if for any upper bound P and any execution prefix e in which only agents from a subset $S \subset \mathcal{A}$ (including BST) interact, the (standard) projection[2] $e|_S$ of e on the agents of S is an execution prefix of $\mathcal{PP}$ for any bound P' such that $|S| \leq P' < P$.

▶ Remark. Similarly, if e is an execution prefix of a semi-uniform protocol $\mathcal{PP}$ for $n' \leq P'$, it is also an execution prefix of $\mathcal{PP}$ for n s.t. $n' \leq n \leq P$ and $P' \leq P$, if we consider e as an execution prefix for p, and then extend the configurations of e with $n - n'$ agents (missing in e and performing no interactions in the extended prefix).

3 Time and Space Optimal Counting under Probabilistic Fairness

3.1 Time Lower Bound for a Space Optimal Protocol

Defining time optimality for a counting protocol asks to be cautious. Indeed, a protocol could be efficient for counting some set of agents and slow for counting others. Think of a protocol that "guesses" initially a count and checks afterwards whether this count is correct or not. On the right set of agents, this counting protocol would converge in zero time. For other sets it is certainly less efficient than the protocols which estimate the count gradually, starting from 0. We would like to avoid such behavior and thus restrict our attention to protocols having always a "proof" that the estimate they have corresponds to a lower bound on the actual population size (i.e., they have observed a sequence of interactions that justifies this count). For such protocols, called here "non-guessing", the estimate of the size (in the variable c) is a non-decreasing function along an execution (and so c is always a lower bound on the number of agents in the population). In the sequel of the section, we consider an arbitrary state-optimal protocol *Count* and we prove that it converges in expected $\Omega(n \log(n))$ interactions with BST, or equivalently $\Omega(n^2 \log(n))$ interactions between agents, or $\Omega(n \log(n)$ parallel time. *Count* uses (optimally) only two states per mobile agent (with one state, counting is impossible [10]). Note also that the result in this section does not assume that *Count* is symmetric.

We call *trace of an execution prefix* the sequence of transitions of BST in this prefix. Thus, a trace T is a sequence of transitions of the form $(s_{BST}, i) \to (s'_{BST}, j)$, where s_{BST} and s'_{BST} are states of BST, and i and j are states of a mobile agent in the corresponding interaction. Note that this sequence captures all the information that BST has, and completely determines its state. Let us denote by $x(T)$ the minimal population size for which there exists at least one execution prefix with trace T. Thus, since the estimate counter c is non-decreasing, we have $c \leq x(T)$: if it was not the case, by definition of $x(T)$, the protocol could run with $x(T)$ agents and estimate, at some point of the execution, that $c > x(T)$; then, to converge, the protocol would be required to decrease c, which is impossible by the assumption above. We denote by $x_i(T)$ the minimal number of mobile agents in state i (w.l.o.g., $i \in \{0, 1\}$) in a configuration at the end of any possible execution prefix corresponding to trace T (and with

[2] The projection of a configuration on a set of agents S is a restriction of the vector representing the configuration to the elements corresponding to the agents of S. Naturally, the projection of an execution e on a set of agents S ($e|_S$) is obtained from e by projecting every configuration of e on S, representing an execution where the agents of $\overline{S}$ (the complement of S) do not interact.

$x(T)$ agents). Obviously, for all execution prefixes with n agents and a trace T, we have $x_0(T) + x_1(T) \leq x(T) \leq n$.

The idea behind Lemma 2 is that, if there is such a transition rule of mobile agents that can decrease the number of agents in state i, then when several such agents are present in the population, making them interact and execute this transition will decrease their number to the minimum. Due to the lack of space, the complete proof is moved to [7].

▶ **Lemma 2.** *If the considered protocol Count has a transition rule that allows to decrease the number of agents in state i through interactions between mobile agents (rules $(i,i) \to (i, 1-i)$, $(i,i) \to (1-i, 1-i)$ or $(i, 1-i) \to (1-i, 1-i)$), then for any trace T, we have $x_i(T) \leq 1$.*

▶ **Lemma 3.** *Let T' be a trace obtained from a trace T by adding a transition $(s_{BST}, i) \to (s'_{BST}, j)$ between BST and a mobile agent. If $x(T') > x(T)$, then $x_{1-i}(T) = x(T)$ and $x(T') = x(T) + 1$.*

Proof. We show the contrapositive: if $x_{1-i}(T) < x(T)$, then $x(T') = x(T)$; if $x_{1-i}(T) = x(T)$, then $x(T') = x(T) + 1$. Let the added interaction in T' be $(s_{BST}, i) \to (s'_{BST}, i)$:

- If $x_{1-i}(T) < x(T)$, some executions with trace T and $x(T)$ agents lead to a configuration with $x(T) - x_{1-i}(T) > 0$ agents in state i. These agents may interact with BST, so that there still exist executions with trace T' and with $x(T)$ agents, and with the same number of agents in both states. Thus $x(T') = x(T)$, $x_0(T') = x_0(T)$, and $x_1(T') = x_1(T)$;
- If $x_{1-i}(T) = x(T)$ (which implies that $x_i(T) = 0$), all executions with trace T and $x(T)$ agents contain no agent in state i. Thus, $x(T') = x(T) + 1$ (it cannot be higher, since one can build an execution with $x(T)$ agents interacting in pattern resulting in trace T, and an extra agent in state i that does not interact, then released for this last interaction). Since, as a result of this interaction, an agent in state i remains, if no rule $(i, 1-i) \to (1-i, 1-i)$ can remove it, $x_i(T) = 1$, else $x_i(T) = 0$. In addition, $x_{1-i}(T') = x_{1-i}(T)$, because the execution described above with $x(T') = x(T) + 1$ agents results in a configuration with $x_{1-i}(T)$ agents in state i. Finally, we have: $x(T') = x(T) + 1$, $x_i(T') \leq 1$, and $x_{1-i}(T') = x_{1-i}(T)$.

Now, let the added interaction in T' be $(s_{BST}, i) \to (s'_{BST}, 1-i)$:

- If $x_{1-i}(T) < x(T)$, some executions with trace T and $x(T)$ agents contain agents in state i. These agents may interact with BST, so that there still exist executions with trace T' and with $x(T)$ agents, and with an agent in state i that has changed its state. Thus $x(T') = x(T)$, $x_i(T') = \max\{x_i(T) - 1, 0\}$, and $x_{1-i}(T') \leq x_{1-i}(T) + 1$.
- If $x_{1-i}(T) = x(T)$ (which implies that $x_i(T) = 0$), all executions with trace T and $x(T)$ agents contain no agent in state i. Thus, trace T' cannot be achieved with $x(T)$ agents, and $x(T') \geq x(T) + 1$. T' can be achieved by adding an extra agent in state i during trace T and releasing it to realize the last interaction, so that $x(T') = x(T) + 1$. An agent in state i has its state changed to $1 - i$, so that $x_i(T') = \max\{x_i(T) - 1, 0\}$, and $x_{1-i}(T') \leq x_{1-i}(T) + 1$.

Thus, x can increase only as the result of an interaction of BST with an agent in state i such that $x_{1-i} = x$. After this interaction, to increment x again, BST must increment x_{1-i}, which it can do only by switching an agent state to $1 - i$. ◀

In particular, this implies that, if interactions between mobile agents can decrease the number of agents in both states 0 and 1, $x_i \leq 1$ for $i \in \{0, 1\}$, and $x \leq 2$, i.e., any trace can be obtained with two agents only, and *Count* is incorrect.

▶ **Theorem 4.** *A two-state non-guessing counting protocol Count (correct under probabilistic fairness) converges in $\Omega(n \log n)$ expected interactions with BST (equivalently, in $\Omega(n \log n)$ expected parallel time).*

Proof. Consider a converging execution of this protocol with $n \geq 3$ mobile agents. Denote by T the trace of this execution until $x(T) = n$ (which occurs, since the protocol converges), and by T' the trace obtained from T by removing its last interaction. Denote by i the mobile agents state such that $x_i(T') = x(T') = n - 1 > 1$ (this state exists by Lemma 3).

Thus, x_i must increase from 0 to n. Now, x_i increases only when BST meets an agent in state $1 - i$ and changes its state to i, and the number of mobile agents in state $1 - i$ can only increase in an interaction with BST (from Lemma 2, as $x_i(T') > 1$, no interaction between mobile agents can increase the number of agents in state $1-i$). Thus, in the last configuration before convergence with $x_i = 0$, all agents are in state $1 - i$ (because x_i eventually reaches n, and can increase only when a mobile agent state is switched from $1 - i$ to i).

Thus, in any converging execution, there is a configuration in which all agents are in state $1 - i$, and then all agents are switched to state i in interactions with BST (except possibly one, that has been counted by BST, and will be switched to state i by it only in a further interaction). Now, if k agents are in state i, the probability for BST to meet an agent in state $1 - i$ in its next interaction is $\frac{n-k}{n}$, and the expected number of interactions (involving BST) before meeting an agent in state i and incrementing x is $\frac{n}{n-k}$.

So, the expected length of an execution before convergence is $\geq \sum_{k=0}^{n-1} \frac{n}{n-k} = \sum_{l=1}^{n} \frac{n}{l} = n \sum_{l=1}^{n} \frac{1}{l} = nH_n$, with H_n the nth harmonic number. It is known that $H_n = \Theta(\log n)$, hence the result. Given that there are n agents in the system, one interaction out of n involves BST in average. Hence, the expected $\Omega(n \log n)$ interactions with BST are equivalent to expected $\Omega(n^2 \log n)$ interactions between agents, and to the expected $\Omega(n \log n)$ parallel time. ◀

3.2 Time and Space Optimal Protocol (Prot. 1)

The time complexity analysis of the one bit space optimal protocol of [10] is presented in [7] and gives the average convergence time of $\Theta(2^n)$ parallel interactions. In this section, we modify this protocol to obtain an (asymptotically) time optimal protocol, Prot. 1, converging in $O(n \log n)$ time, and still optimally using only one bit of memory per mobile agent. We present and prove this protocol and its convergence time below.

In Prot. 1, each mobile agent can be in one of two states 0 or 1, and respectively called 0 or 1 agent. We write c_0 and c_1 for the protocol's count of 0 and 1 agents resp., and n_0 and n_1 for the actual number of 0 and 1 agents in the population. The total number of agents is then $n = n_0 + n_1$ and the base station's estimate of n is $c = c_0 + c_1$. The values c_0 and c_1 are both initialized to 0. They may be seen as the implementations of the x_0 and x_1 used in the lower bound proof in the previous subsection.

The modified protocol proceeds in alternating phases. In a **zero phase**, BST only converts zeros to ones. Whenever it does so, it decrements c_0 if it is positive and increments c_1. In a **one phase**, it does the reverse. We start the protocol in a zero phase.

The same argument as for the original protocol shows that $c_b \leq n_b$ holds as an invariant and that $c = c_0 + c_1$ is non-decreasing over time (Lemma 1 in [10]). If, in addition, we can stay in two phases long enough that every agent is converted from b to $1 - b$ in the first phase, and then every agent is converted from $1 - b$ to b in the second phase, at the end of the second phase we will have $c = n$, giving convergence.

Let us now specify when BST switches between phases. Suppose that BST is going to start in a b phase. We adopt the following procedure (in two stages):

Protocol 1 – Time and Space Optimal Counting under Probabilistic Fairness.

Variables at BST:
c_0: non-negative integer, initialized to 0; eventually holds n_0
c_1: non-negative integer, initialized to 0; eventually holds n_1
c: non-negative integer initialized to 0; eventually holds n
cnt: non-negative integer initialized to 0
$phase \in \{0,1\}$, initialized to 0
Variable at a mobile agent x:
$b \in \{0,1\}$, initialized *arbitrarily*

```
1:  when a mobile agent x with mark b interacts with BST do
2:    if b = phase then
3:      cnt ← 0
4:      if c_b > 0 then
5:        c_b ← c_b − 1
6:      b ← 1 − b
7:      c_b ← c_b + 1
8:    else if cnt ≥ 6(c_b ln c_b + 1) then
9:      cnt ← 0
10:     phase ← 1 − phase
11:   else if c_phase = 0 then
12:     cnt ← cnt + 1
13:   c ← c_0 + c_1
```

1. (*pre-phase*) Flip any b agents we encounter to $1-b$ as long as $c_b > 0$.
2. (*the phase itself*) Continuing flipping any b agents BST encounters to $1-b$ until it sees $6(c_b \ln c_b + 1)$ agents marked $1-b$ in a row without seeing an agent marked b. If this event occurs, flip the phase (switch to the $1-b$ phase).

The first rule (the pre-phase) guarantees that whenever we start a b phase, c_{1-b} is always zero. This in turn guarantees that c_b is never lower at the start of a b phase than it is at the start of any previous b phase.

For the convergence bound, begin by bounding the likely length of a phase:

▶ **Lemma 5.** *Each phase requires $O(n \log n)$ parallel interactions with high probability.*

Proof. Suppose we are in a b phase. Using standard bounds on the Coupon Collector Problem, it holds with high probability that BST has interacted with every agent after $O(n \log n)$ interactions. So either the phase has already ended, or every agent now carries $1-b$. In the latter case, the phase can run for at most $6(n \ln n + 1) = O(n \log n)$ interactions before ending. ◀

We now show that, on average, the protocol executes $O(1)$ phases. This requires the following technical lemma showing that BST finds all b agents in a b phase if there are enough to begin with.

▶ **Lemma 6.** *If phase b starts with $n_b \geq n/2$, then it ends with $n_b = 0$ with probability at least $1/2$.*

Proof. For simplicity we will assume $b = 0$; the $b = 1$ case is symmetric. So we are looking at a zero phase that starts with $n_0 \geq n/2$. From the structure of the protocol, we know that at the start of this phase, $c_1 = 0$, but c_0 might be larger. It happens that the worst case is when $c_0 = 0$, but we will analyze the process for any initial value of c_0.

In the analysis below we will fix n_0, n_1, to their values at the start of the phase. To keep track of what happens, let i be the number of zero values converted to ones so far during this phase; given the value of i, this gives $n_0 - i$ zeros and $n_1 + i$ ones in the population, and the value of the c_1 register will be i. We fail to convert all zeros to ones if we exit the phase while i is less than n_0.

For each particular value of i, this occurs only if (a) c_0 is already 0, and (b) BST observes $6(n \ln n + 1)$ ones in a row. Whether or not $c_0 = 0$, the latter event occurs with probability exactly

$$\left(\frac{n_1 + i}{n}\right)^{6(i \ln i + 1)} \tag{1}$$

which by the union bound gives an upper bound on the probability that we leave the phase for any $i < n_0$ of

$$\sum_{i=0}^{n_0 - 1} \left(\frac{n_1 + i}{n}\right)^{6(i \ln i + 1)} \tag{2}$$

We will bound this sum by considering the terms with $i < n_0/2$ and $i \geq n_0/2$ separately. The detailed computations for each case appear in [7] and give a bound of $2/5$ for the case $i < n_0/2$, and $1/200$ for $i \geq n_0/2$. The original sum is thus bounded by $2/5 + 1/200 < 1/2$ for all $n > 0$, giving the claimed bound. ◀

▶ **Theorem 7.** *The modified protocol (Prot. 1) converges to $c = n$ in an expected $O(n \log n)$ interactions with BST (equivalently, in an expected $O(n \log n)$ parallel interactions).*

Proof. There are two cases, depending on the initial value of n_0.

1. If $n_0 \geq n/2$ in the starting configuration, then $n_0 \geq n/2$ at the start of each zero phase. From Lemma 6, BST converts all zeros to ones in any of these phases with probability at least $1/2$. If this event occurs, the following one phase converts all ones to zeros with probability at least $1/2$ as well, giving a probability of at least $1/4$ for each pair of phases that we converge to the correct count. Thus the protocol converges in an expected $4 \cdot 2 = 8$ phases.
2. If $n_0 < n/2$, then the initial zero phase ends with at least $n/2$ ones (because any conversion during this phase can only increase the number of ones). So the first one phase starts with $n_1 = n/2$. Repeating the above analysis shows that the protocol converges after at most 8 phases on average on top of the initial zero phase, giving an expected 9 phases total.

Because each of these $O(1)$ phases takes $O(n \log n)$ expected interactions (Lemma 5), this gives a total expected number ◀

4 Time Lower Bound for Space Optimal Counting under Weak Fairness

To obtain this lower bound we first prove properties that have to be satisfied by any space optimal symmetric counting protocol functioning under weak fairness. These properties are

important by themselves, as they can be useful in future studies of counting under weak fairness in PP. For instance, Proposition 8, states that a counting protocol has to distinctly name all the agents in any population of size $n < P$. Recall that P is the upper bound on the size n of the population.

Next, from Proposition 8 and Lemma 9, it easily follows that any symmetric counting protocol under weak fairness has to use at least P states per mobile agent (to be able to count any population of at most P agents). This gives a somewhat simpler proof than the original one in [11].

The next important property, given in Proposition 11, is that any space optimal symmetric counting protocol under weak fairness has a unique "sink" state m s.t., for every possible state $s \in Q$ of a mobile agent, there is a transition sequence $(s, s) \xrightarrow{*} (m, m)$, with $(m, m) \rightarrow (m, m)$ and m cannot be one of the distinct names given by the protocol in case $n < P$.

The results above show in particular that, for any considered space-optimal counting protocol, if mobile agents are not named yet, agents in state m will continue to appear (for any P and $n \leq P$).

Moreover, recall that we consider counting with non-initialized mobile agents. In this case, to overcome the known impossibility [11], we assume one initialized and distinguishable agent BST that eventually counts the other n (mobile) agents. Note that having a distinguishable agent is necessary. To see this, consider a starting configuration where all agents start at the same state (which has to be equal to the state of the only initialized agent). If the size of the population is even, by Proposition 11, there is a weakly fair execution reaching and staying in the configuration where all agents are in the "sink" state m, and by Proposition 8, no counting can be realized.

Using the above-mentioned properties and those of semi-uniform protocols (see Definition 1 in Sec. 2), we prove the lower bound given in Theorem 14. This is one of the main results of the paper. It shows that, under weak fairness, counting undistinguishable, state-optimal and non-initialized agents in symmetric PP is a costly task in terms of a convergence time. It takes $\Omega(2^n)$ non-null transitions.

Finally, we show (Proposition 15 in [7]) that the space optimal protocol for weak fairness presented in [10] converges in $\Theta(2^n)$ non-null transitions. This proves that this is a time optimal protocol among all the space optimal semi-uniform protocols, under weak fairness.

▶ **Proposition 8.** *Let Count be a (silent or not) counting protocol correct under weak fairness (for any $n \leq P$). For any weakly fair execution $e = C_1, C_2, C_3, \ldots, C_j, \ldots$ of Count on a population $\mathcal{A}$ of size $n < P$, there is an integer k such that, for any $j \geq k$, no two mobile agents are in the same state in C_j.*

Proof. Let us assume, by contradiction, that in e, there are infinitely many configurations with two agents in the same state. Since the state space is finite and the number of agents too, two specific agents x_2 and x_3 from $\mathcal{A}$ are necessarily simultaneously in some state $s \in Q$ in infinitely many configurations. Let $C_{j_1}, C_{j_2}, C_{j_3}, \ldots$ be these configurations such that $e = e_1, C_{j_1}, e_2, C_{j_2}, e_3, C_{j_3}, \ldots$ W.l.o.g., we choose these configurations such that, in every execution segment e_i, every agent in $\mathcal{A}$ interacts with every other (this is possible with weak fairness).

Now consider a population $\mathcal{A}' = \mathcal{A} \cup \{x_1\}$ of size $n + 1$. To prove the proposition, we will construct a weakly fair execution e' of *Count* in population $\mathcal{A}'$ where no agent can distinguish e' from e, and where consequently *Count* wrongly counts only n agents instead of the existing $n + 1$.

We construct e' based on e. First, we assume that in e', x_1 is in state s in the starting configuration, and $e' = e'_1, C'_{j_1}, e'_2, C'_{j_2}, e'_3, C'_{j_3}, \ldots$ such that each segment e'_i follows the same

transition sequence as in e_i, but where the agents x_2 or x_3 participating in the corresponding interactions can be replaced by x_1 in the appropriate state, as we explain below. We ensure also that in every C'_{j_i}, each of the three agents x_1, x_2, x_3 is in state s.

More precisely, in segment $e'_{3r+1}, C'_{j_{3r+1}}$ $(r \geq 0)$, agent x_1 does not interact with the rest of the agents, and all the others interact exactly as in $e_{3r+1}, C_{j_{3r+1}}$ (each agent x_i is in state s in $C'_{j_{3r+1}}$). In $e'_{3r+2}, C'_{j_{3r+2}}$, agent x_2 does not interact with the rest of the agents, and x_1 replaces x_2 in all the interactions where x_2 interacts in $e_{3r+2}, C_{j_{3r+2}}$, and all the others interact as in e_{3r+2} (but with x_1 instead of x_2 in the corresponding interactions). Each agent x_i is in state s in $C'_{j_{3r+2}}$. In $e'_{3r+3}, C'_{j_{3r+3}}$, agent x_3 does not interact with the rest of the agents, and x_1 replaces x_3 in all the interactions where x_3 interacts in $e_{3r+3}, C_{j_{3r+3}}$, and all the others interact as in e_{3r+3} (but with x_1 instead of x_3 in the corresponding interactions). Each agent x_i is in state s in $C'_{j_{3r+3}}$.

We emphasize again that e' is possible, because in every C'_{j_i}, each of the three agents x_1, x_2, x_3 is in state s, so any of them can replace any other in the transition sequence of the next segment e_{i+1}. Moreover, e' is weakly fair, because each agent x_i interacts with all the other agents in the appropriate e'_i segments (and by the assumption on e_i), and other agents too, due to the weak fairness of e. Finally, in e', every agent from $\mathcal{A}$ (including BST), executes exactly the same sequence of transition rules as it does in e, so no agent can distinguish the fact that the population is actually $\mathcal{A}'$ with $n+1$ agents, and *Count* counts only n agents as it does in e. This is a contradiction to the assumption that *Count* is a correct counting protocol. ◀

The proof of Lemma 9 uses similar techniques as the proof of Proposition 8 and appears in [7].

▶ **Lemma 9.** *Let Count be a symmetric (silent or not) counting protocol correct under weak fairness (for any $n \leq P$). Consider any weakly fair execution $e = C_1, C_2, C_3, \ldots, C_j, \ldots$ of Count on a population $\mathcal{A}$ of size $n < P$. There is an integer k such that, for any $j \geq k$, no mobile agent is in a state $m \in Q$ such that there is a possible sequence of transitions of Count $(m, m) \xrightarrow{*} (m, m)$.*

The following two propositions follow from Proposition 8 and Lemma 9. A proof of Proposition 10 uses similar techniques as the proof of Proposition 11 and appears in [7].

▶ **Proposition 10.** *Any symmetric counting protocol Count correct for any $n \leq P$ (undistinguishable and non-initialized) mobile agents, under weak fairness, has to use at least P states per mobile agent.*

▶ **Proposition 11.** *Consider any symmetric (silent or not) counting protocol Count correct under weak fairness (for any $n \leq P$), and using at most P states per mobile agent. For every state $s \in Q$, there is a transition sequence $(s, s) \xrightarrow{*} (m, m)$, s.t. m is unique and does not appear infinitely often in executions with $n < P$. Moreover, $(m, m) \rightarrow (m, m)$.*

Proof. As *Count* is symmetric, any two agents, both in some state $s \in Q$, in an interaction, have to execute a symmetric transition of the form $(s, s) \rightarrow (s_1, s_1)$. Thus there is a possible sequence of transitions $(s, s) \rightarrow (s_1, s_1) \rightarrow (s_2, s_2) \rightarrow (s_3, s_3) \ldots$ As mobile agents are finite state, for some $j > i \geq 1$, $s_i = s_j$, i.e. $(s_i, s_i) \xrightarrow{*} (s_i, s_i)$. By Lemma 9, $s_i = m$ s.t. m does not appear infinitely often in executions with $n < P$. As there are at least $P - 1$ states appearing infinitely often in an execution with $n = P - 1$ (by Proposition 8), there is at most one such possible state m in a P state protocol. Thus, the first part of the lemma holds.

Finally, by contradiction, if $(m, m) \rightarrow (s, s)$ s.t. $s \neq m$, then the previous part of the proof implies $(s, s) \xrightarrow{*} (s, s)$. When $n = P - 1$, and as *Count* uses only P states, and m never

appears infinitely often in an execution, s does appear infinitely often in configurations of an execution (by Proposition 8). This is a contradiction to Lemma 9. Thus, $(m, m) \not\to (m, m)$. ◀

To prove the lower bound (Theorem 14), in addition to the results above, we use the following definitions related to the considered counting protocols.

▶ **Definition 12.**

- We call *homonyms*, or *homonymous agents*, mobile agents in the population having the same state, but different from m.
- We say that two (or more) homonyms (in state s) are *reduced (to m)* whenever a sequence of transitions $(s, s) \xrightarrow{*} (m, m)$ is applied to them.
- We say that a mobile agent is *named* if it has a state different from m (a *name*). A group of agents is *named* if each of them is named with a distinct name. Similarly, a configuration of agents is named, if all the agents in this configuration are named.
- A *reduced (from homonyms) configuration* is a configuration without any homonym. Given a configuration C, let $R(C)$ be the set of names of mobile agents appearing an odd number of times in C, i.e., the set of names appearing in the corresponding reduced configuration.
- For any two sets $E, E' \subseteq \{1, \ldots, n\}$, we denote by $E \triangle E' \equiv E \cup E' - E \cap E'$ their symmetric difference. In particular, $E \triangle \{e\}$ $(e \in \{1, \ldots, n\})$ is $E \cup \{e\}$ if $e \notin E$, and $E - \{e\}$ if $e \in E$.
- A *stationary point (or state) of BST* is a state s_{BST} of BST such that $(s_{BST}, m) \to (s'_{BST}, m)$ and s'_{BST} is also stationary. (Note that this transition sequence can be broken, i.e., BST can change its state to a non-stationary one, after an interaction with an agent in a state $s \neq m$.)
- The *naming sequence* is a sequence $U = (s^i_{BST}, s_i)_{i \geq 1}$ of pairs (s^i_{BST}, s_i), where s^i_{BST} is a BST state and s_i is a mobile agent state. U is defined inductively as follows: s^1_{BST} is the initial state of BST, and for any $i \geq 1$, $(s^i_{BST}, m) \to (s^{i+1}_{BST}, s_i)$ is a transition rule of the protocol. Let U_j be a prefix of U s.t. $U_j = (s^i_{BST}, s_i)_{1 \leq i \leq j}$.

To obtain the result of Theorem 14, we focus on the set of the longest execution prefixes where BST meets and names agents in state m (according to the fixed naming sequence, defined in Definition 12). In such prefixes, we study the possibility of the occurrence of a stationary point (Definition 12). We show that, for a semi-uniform counting protocol (Definition 1, Sec. 2), for any $n < P$, such a point does not exist (Lemma 13), before BST has named all the n agents. That is, in the case of the considered executions, BST will continue giving names to m-state agents that it meets (it won't be "blocked" waiting for some named agent, for possibly deciding to change its strategy accordingly).

So, using Lemma 13, we prove that there exists an execution prefix in which BST continuously meets and names agents in state m without entering a stationary state. Then, we show that the longest such prefix corresponds to the naming sequence of length $\Omega(2^n)$. This is by observing that the number of starting unnamed (reduced) configurations is $2^n - 1$, for $n = P - 1$, and that after each step (transition) of the naming sequence, BST can accomplish naming of only one such starting configuration. To prove the result for any n and P, we use Definition 1 of semi-uniform protocols. Intuitively, for such a protocol, when only a subset of a population of $x < n$ agents interacts with BST, BST should behave like $P = x$ is possible, even if it is larger.

In the following lemma, we show that the naming sequence does not contain any stationary state.

▶ **Lemma 13.** *Let Count be a symmetric (silent or not) semi-uniform counting protocol correct under weak fairness (for any $n \leq P$) and using P states per mobile agent. The naming sequence U (Definition 12) of Count does not contain any stationary state.*

Proof. Assume by contradiction that the first stationary state in $U = (s^i_{BST}, s_i)_{i\geq 1}$ is in the k^{th} step (element), i.e., the state s^k_{BST} is stationary, for some $k > 0$. In the following, we assume that $P > 2k$ (we justify this assumption later) and for any $j \geq 0$, we construct an execution e_j of *Count* for a population of $2k + j$ ($< P$) agents. In e_j, $k + j$ agents are initially in state m, while the k other agents are in states s_1, s_2, ..., s_k (these are the states s_i appearing in U). Each e_j is composed of two phases.

In the first phase, let k m-state agents interact one by one with BST, and at the end of this phase (by the definition of U) the states of these agents are s_1, s_2, ..., s_k. Notice that now among these k agents (call it the first sub-population), those that have names have homonyms in the second sub-population (of other $k + j$ agents). Notice also that, by Definition 1 of semi-uniform protocols, this is also a prefix of an execution projected on a population of only k agents. By the same property, this can be also a prefix of execution for a larger population and for an upper bound P as large as we want. Thus, $P > 2k$ is a valid assumption.

In the second phase, let every named (let us say, by s) agent of the first sub-population interact with its homonym in the second sub-population, s.t. the sequence of transitions $(s, s) \xrightarrow{*} (m, m)$ takes place for each such pair of homonyms (possible by Proposition 11). At the end of this second phase, all agents are thus in state m (at most $2k$ were homonyms at the end of the first phase, and j were initially in state m and never interacted).

Now, no matter how the agents continue to interact, as all mobile agents are in state m, and BST is in a stationary state s^k_{BS}, all mobile agents remain forever in state m (Proposition 11) and BST remains in a stationary state (by definition of such a state). Thus, after the second phase, we make all pairs of agents interact infinitely often to obtain a weakly fair execution for any j. In all these constructed executions, no agent can distinguish between the executions and the corresponding population sizes ($2k + j < P$ agents for any $j \geq 0$), and thus a correct counting cannot be obtained.

By Definition 1 of a semi-uniform protocol, the projection of the first phase (of any e_j) on the first sub-population (a group of k agents interacting with BST in this phase), is also a prefix of an execution of *Count* for a bound $P' \geq k$ ($k \leq P' \leq P$). Thus, for any such k and P', there is no stationary point in U_k. In other words, the lemma holds for any value of k and P, and thus also for any prefix of U. ◀

▶ **Theorem 14.** *Let Count be a symmetric (silent or not) semi-uniform counting protocol correct under weak fairness (for any $n \leq P$) and using P states per mobile agent. The convergence time of Count is at least $2^n - 1$ non-null transitions.*

Proof. We will build an execution of *Count* where the length of the corresponding naming sequence U before convergence (and thus the convergence time of *Count*) is at least $2^n - 1$.

Consider a population of $n = P - 1$ agents. Consider a possible execution prefix e where BST interacts only with m-state agents as long as they are not distinctly named. If the agents are not distinctly named, by Proposition 11, there is always either at least one agent in state m, or some homonyms that can be reduced to m. So, assume that in e, whenever the mobile agents are not distinctly named and there is no agent in state m, a reduction of some homonyms is done. Then, an agent in state m interacts with BST. By Lemma 13, in every such corresponding transition, the state of BST is not stationary, and thus eventually

an m-state mobile agent is "given" a name by BST. Let us repeat this scenario, until all the n agents are named. This is an execution prefix e (with $n = P - 1$) that we consider below.

By Lemma 13, *Count* has to name $n = P - 1$ mobile agents, starting from any unnamed configuration C_j. For this specific case of $n = P-1$, there is *exactly one* possible configuration C^* (ignoring the state of BST and the permuted configurations[3]) where all n mobile agents are named. Notice that $R(C^*) = Q \setminus m$ ($R()$ is defined in Definition 12). Consider a prefix $U_j = (s^i_{BS}, s_i)_{0 \leq i \leq j}$ of the unique naming sequence U. Notice also that, by Definition 12, $A \triangle B = A' \triangle B$ iff $A = A'$. This implies that, given U_j, there is a unique $R(C_j)$ such that $R(C_j) \triangle \{s_1\} \triangle \{s_2\} \triangle \{s_3\} \dots \triangle \{s_j\} = \{a_1, a_2, \dots, a_{P-1}\} = R(C^*) = Q \setminus m$. As $|\{R(C_j) : C_j \text{ is any possible unnamed starting configuration}\}| = 2^{P-1} - 1$, the length of U is at least $2^{P-1} - 1 = 2^n - 1$. Hence, the length of the longest execution e is at least $2^{n-1} - 1$ s.t. $n = P - 1$. By the Remark following Definition 1, e is also an execution prefix for any bound $\geq P$, given the same population size. Thus, the theorem actually holds for any n and any upper bound P on n. ◀

5 Conclusion and Perspectives

This work is a sequel to [10] and it answers the questions concerning time complexity of the symmetric space optimal protocols proposed there. What can be learned from the current work is that there exists a big difference, not only in terms of the required space, but also in terms of time complexity, between the case where the interactions between agents are random and the case where they are only weakly fair.

From a more practical point of view, it would be interesting to investigate if this difference still exists when more memory space is given to the agents. We already know that, concerning weak fairness, a supplementary bit ($2P$ states) allows to design protocols like in [20] with a logarithmic round complexity (a round being a shortest fragment of execution where each agent interacts with each other), while another additional bit allows to solve this problem in only constant number of rounds [11]. Concerning global or probabilistic fairness, there exist less studies about counting protocols and especially about their complexity analysis. For example, it would be certainly interesting to determine which size of memory is needed, for having an expected constant convergence time. More generally, studying formally the trade-offs between space and time complexities for counting algorithms in population protocols could be a valuable sequel to the present work.

References

1 D. Alistarh, J. Aspnes, D. Eisenstat, R. Gelashvili, and R. L. Rivest. Time-space trade-offs in population protocols. *CoRR*, abs/1602.08032, 2016. URL: `http://arxiv.org/abs/1602.08032`.

2 D. Alistarh and R. Gelashvili. Polylogarithmic-time leader election in population protocols. In *ICALP*, pages 479–491, 2015. `doi:10.1007/978-3-662-47666-6_38`.

3 D. Alistarh, R. Gelashvili, and M. Vojnovic. Fast and exact majority in population protocols. In *PODC*, pages 47–56, 2015. `doi:10.1145/2767386.2767429`.

4 D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Dist. Comp.*, 18(4):235–253, 2006. `doi:10.1007/s00446-005-0138-3`.

[3] A permuted configuration obtained by permuting the elements in the configuration vector.

5 D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Dist. Comp.*, 20(4):279–304, 2007. doi:10.1007/s00446-007-0040-2.

6 A. F. Anta, M. A. Mosteiro, and C. Thraves. An early-stopping protocol for computing aggregate functions in sensor networks. *J. Parallel Distrib. Comput.*, 73(2):111–121, 2013. doi:10.1016/j.jpdc.2012.09.013.

7 J. Aspnes, J. Beauquier, J. Burman, and D. Sohier. Time and Space Optimal Counting in Population Protocols, https://hal.inria.fr/hal-01399797. Technical report, Yale, Nov 2016. URL: https://hal.inria.fr/hal-01399797.

8 C. Baquero, P. S. Almeida, R. Menezes, and P. Jesus. Extrema propagation: Fast distributed estimation of sums and network sizes. *IEEE Trans. Parallel Distrib. Syst.*, 23(4):668–675, 2012. doi:10.1109/TPDS.2011.209.

9 J. Beauquier, J. Burman, and S. Clavière. Comptage et nommage simples et efficaces dans les protocoles de populations symétriques. In *ALGOTEL 2014*, pages 1–4, Jun 2014. URL: https://hal.archives-ouvertes.fr/hal-00986109.

10 J. Beauquier, J. Burman, S. Clavière, and D. Sohier. Space-optimal counting in population protocols. In *DISC*, pages 631–646, 2015. doi:10.1007/978-3-662-48653-5_42.

11 J. Beauquier, J. Clement, S. Messika, L. Rosaz, and B. Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *DISC*, pages 63–76, 2007.

12 G. Di Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. Conscious and unconscious counting on anonymous dynamic networks. In *ICDCN*, pages 257–271, 2014. doi:10.1007/978-3-642-45249-9_17.

13 G. Di Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. Counting in anonymous dynamic networks under worst-case adversary. In *ICDCS*, pages 338–347, 2014. doi:10.1109/ICDCS.2014.42.

14 G. A. Di Luna and R. Baldoni. Brief announcement: Investigating the cost of anonymity on dynamic networks. In *PODC*, pages 339–341, 2015. doi:10.1145/2767386.2767442.

15 S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999. URL: http://link.springer.de/link/service/journals/00236/bibs/9036006/90360447.htm.

16 S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Dist. Comp.*, 7(1):3–16, 1993. doi:10.1007/BF02278851.

17 D. Doty and D. Soloveichik. Stable leader election in population protocols requires linear time. In *DISC*, pages 602–616, 2015. doi:10.1007/978-3-662-48653-5_40.

18 A. J. Ganesh, A.-M. Kermarrec, E. Le Merrer, and L. Massoulié. Peer counting and sampling in overlay networks based on random walks. *Dist. Comp.*, 20(4):267–278, 2007. doi:10.1007/s00446-007-0027-z.

19 C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Perform. Eval.*, 63(3):241–263, 2006. doi:10.1016/j.peva.2005.01.002.

20 T. Izumi, K. Kinpara, T. Izumi, and K. Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theor. Comput. Sci.*, 552:99–108, 2014. doi:10.1016/j.tcs.2014.07.028.

21 H. Jiang. *Distributed Systems of Simple Interacting Agents*. PhD thesis, Yale University, 2007.

22 P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS*, pages 96–107, 2002. doi:10.1145/605397.605408.

23 D. Kostoulas, D. Psaltoulis, I. Gupta, K. P. Birman, and A. J. Demers. Active and passive techniques for group size estimation in large-scale and dynamic distributed systems. *Journal of Systems and Software*, 80(10):1639–1658, 2007. doi:10.1016/j.jss.2007.01.014.

24 S. Lahde, M. Doering, W. Pöttner, G. Lammert, and L. C. Wolf. A practical analysis of communication characteristics for mobile and distributed pollution measurements on the road. *Wirel. Comm. and Mob. Comput.*, 7(10):1209–1218, 2007.

25 O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Naming and counting in anonymous unknown dynamic networks. In *SSS*, pages 281–295, 2013. doi:10.1007/978-3-319-03089-0_20.

26 A. Milani and M. A. Mosteiro. A faster counting protocol for anonymous dynamic networks. In *OPODIS*, pages 28:1–28:13, 2015. doi:10.4230/LIPIcs.OPODIS.2015.28.

27 Y. Mocquard, E. Anceaume, J. Aspnes, Y. Busnel, and B. Sericola. Counting with population protocols. In *NCA*, pages 35–42, 2015. doi:10.1109/NCA.2015.35.

28 Pigeon Air Patrol project, 2016. URL: http://www.pigeonairpatrol.com/.

29 B. F. Ribeiro and D. F. Towsley. Estimating and sampling graphs with multidimensional random walks. In *ACM SIGCOMM*, pages 390–403, 2010. doi:10.1145/1879141.1879192.

30 G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000. 2nd ed.

31 D. Varagnolo, G. Pillonetto, and L. Schenato. Distributed statistical estimation of the number of nodes in sensor networks. In *CDC*, pages 1498–1503, 2010. doi:10.1109/CDC.2010.5717355.

Deterministic Population Protocols for Exact Majority and Plurality*

Leszek Gąsieniec[1], David Hamilton[2], Russell Martin[3], Paul G. Spirakis[4], and Grzegorz Stachowiak[5]

1 **Department of Computer Science, University of Liverpool, Liverpool, UK**
L.A.Gasieniec@liverpool.ac.uk
2 **Department of Computer Science, University of Liverpool, Liverpool, UK**
D.D.Hamilton@liverpool.ac.uk
3 **Department of Computer Science, University of Liverpool, Liverpool, UK**
Russell.Martin@liverpool.ac.uk
4 **Department of Computer Science, University of Liverpool, Liverpool, UK**
P.Spirakis@liverpool.ac.uk
5 **Instytut Informatyki, Uniwersytet Wrocławski, Wrocław, Poland**
gst@cs.uni.wroc.pl

Abstract

In this paper we study space-efficient deterministic population protocols for several variants of the *majority* problem including *plurality consensus*. We focus on space efficient majority protocols in populations with an arbitrary number of colours C represented by k-bit labels, where $k = \lceil \log C \rceil$. In particular, we present asymptotically space-optimal (with respect to the adopted k-bit representation of colours) protocols for (1) the *absolute majority* problem, i.e., a protocol which decides whether a single colour dominates all other colours considered together, and (2) the *relative majority problem*, also known in the literature as *plurality consensus*, in which colours declare their volume superiority versus other individual colours.

The new population protocols proposed in this paper rely on a *dynamic formulation* of the majority problem in which the colours originally present in the population can be changed by an *external force* during the communication process. The considered dynamic formulation is based on the concepts studied in [4] and [24] about *stabilizing inputs* and *composition of population protocols*. Also, the protocols presented in this paper use a composition of some known protocols for static and dynamic majority.

1998 ACM Subject Classification G.2.1 Combinatorial Algorithms, G.2.2 Network Problems

Keywords and phrases Deterministic population protocols, majority, plurality consenus

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.14

1 Introduction

The model of *population protocols* adopted in this paper was proposed first in the seminal paper by Angluin *et al.* [5] and popularised later in [7]. Their model provides a suitable theoretical framework for studying pairwise interactions within a large collection of anonymous (indistinguishable) *entities*, also referred to as *agents*, equipped with little computational power. The entities are modelled as *finite state machines*. When two entities engage in

* This work is sponsored in part by the University of Liverpool initiative Networks Sciences and Technologies (NeST) and by the Polish National Science Centre grant DEC-2012/06/M/ST6/00459.

interaction they mutually access their local states and, on the conclusion of the encounter, their states get updated according to the global (shared) *transition function*. In the *asynchronous model*, also adopted in this paper, the order of interactions in consecutive rounds is unpredictable but fair, i.e., none of the pairs of entities can be starved from interaction. In this model, the main emphasis is on feasibility of the solution, subject to the limit on the number of states available to the entities. In the *probabilistic model*, in each round the *random scheduler* picks a pair of entities uniformly at random. In the presence of the *random scheduler*, on the top of space restrictions, one is also interested in the time complexity of a specific distributed task. A population protocol terminates if all participating entities eventually agree on some value represented by dedicated states, independently of the order of interactions. This value can reflect the colour or the size of selected majority [6, 7, 9, 23], the identity of the leader [2, 1, 19], but also completion of more complex tasks such as network formation [25], counting [26], and others.

In this paper, the adopted computation model encompasses a population A of n fault-free entities, each equipped with a $O(k)$-bit memory, where 2^k is the bound on the number C of colours present in the population. This is in contrast to the majority settings considered earlier in [6, 7, 23, 9] where only two original colours were permitted. Here each entity is coloured with exactly one of C available colours and a k-bit label representing this colour is kept in the entity's memory.

As indicated before, the entities communicate in pairs in an *asynchronous manner*. The main task in the *majority problem* is to identify the most frequent colour in the population. Due to presence of more than two colours in the population, we distinguish between the *absolute majority*, i.e., where one colour dominates all others taken together, and the *relative majority*, also known in the literature as *plurality consensus*, where the population is expected to agree on (one of) the most frequent colour(s). We also distinguish between the *static majority* in which the original colours of entities cannot be altered in time – the assumption used in the past work on majority protocols [6, 7, 23, 9], and the *dynamic majority* in which the original colours of entities can be changed in due course by an *external force*, and by doing so may alter the outcome of the majority protocol. This is the main reason why in our model the entities must store their original colour, which could be altered at any time but only by the external force, in addition to $O(k)$ memory bits required during interactions and to report the majority on the conclusion of the computation process.

The model with the external force adopted in this paper was considered earlier in [24] under the name *computing with stabilizing inputs*. Note that the dynamic protocol described in Section 3 is a special variant of self-stabilization, as state alterations done by the external force are permitted only between certain (colour indicating) states. We would also like to emphasise that protocols for absolute majority presented in Section 4 and the relative majority in Section 5 refer to earlier work on *composition of population protocols* from [4].

In our model, entities interact using a classical population protocol, i.e., via global grammars mapping pairs of states to pairs of states. In particular, no exchange of local memories happens during pairwise interactions. The entities use their local memory in order to organise the sub-protocols executed and in order to draw local conclusions. Thus, if we count the states needed for entities' interactions, we require only $\Theta(k)$ states in our algorithms for absolute and relative majority, and only a constant number of states for protocols computing static and dynamic majority of two colours. In addition, we need only $O(k)$ bits of local memory per entity in our absolute and relative majority protocols in order to handle up to 2^k colours, which is optimal in terms of space requirements.

1.1 Related Work

The population protocol model was initially introduced to simulate behaviour of animal populations [5, 6]. In [5] we can find a formal definition of computations in populations where pairwise interactions of finite-state agents advance the computation. The authors showed a fundamental result that any predicate which is semi-liner can be stably computed by such protocols. In the introduction of their paper, they present a protocol for majority which is exactly the same as the protocol in Section 2 of this paper. In [24] the authors present several models of population protocols including protocols in which each entity of the population is allowed to have some memory, and they discuss several classes of computable predicates in those models. In the first pages, they present a protocol for majority as in [5], which is almost the first protocol presented here. We have included this first protocol in this paper because we add a detailed explanation about reporting ties. Self-stabilizing population protocols were defined in [8] and properties of such protocols were demonstrated. Stabilizing population protocols in the presence of faults were considered in [16].

In due course, population protocols proved to be a useful abstraction in diverse environments including, e.g., wireless sensor networks [4, 27, 20], chemical reaction networks [14], and gene regulatory networks [13]. A large portion of work devoted to population protocols refers to the majority problem. In particular, in [7] the authors study populations with entities governed by 3 states and propose a probabilistic population protocol for *approximate majority*, i.e., where the initial difference between the volumes of the two colours does not fall below $\omega(\sqrt{n}\log n)$. The algorithm stabilises in $O(n\log n)$ rounds with high probability. It also tolerates groups of $o(\sqrt{n})$ entities expressing Byzantine behaviour. Further analysis of this protocol and its 4-state amendment leading to the first efficient exact majority protocol can be found in [23]. Another aspect referring to the parallelism of majority population protocols in the presence of a random scheduler has been studied by Alistarh et al. in [3]. They proposed a poly-logarithmic time majority protocol for entities equipped with memories of size $O(1/\varepsilon + \log n \log 1/\varepsilon)$, for any $\varepsilon > 0$. They also study the respective lower bounds. In a very recent work [1] Alistarh et al. consider a wide spectrum of time and space trade-offs for population protocols and they propose a fast Split-Join majority algorithm stabilising in $O(\log^3 n)$ parallel rounds with high probability. An interesting extension of population protocols to the *random walk* model can be found in [9]. Please note that neither of the majority algorithms discussed above is able to report the tie.

The relative majority variant considered in this paper is well known in the literature under the name of *plurality consensus*. In contrast to the deterministic sequential model adopted in this paper, so far plurality consensus was considered solely in the *gossiping model*. In this model, in a sequence of synchronous rounds each entity contacts a random neighbour simultaneously. Moreover, the protocols converge under the assumption that the number of entities supporting the winning colour must exceed those supporting any other colour by a sufficiently large bias. In this model one explores parallelism of connections aiming at protocols stabilising rapidly with high probability. Doerr et al. [17] explored the *power of two choices* in complete graphs, proposing a stabilisation protocol in the binary case requiring constant memory and message size. Their protocol converges in $O(\log n)$ rounds assuming a bias of size $\Omega(\sqrt{n\log n})$. A more rigorous analysis of this protocol can be found in [15], also in networks modeled by regular graphs, for which the authors provide tight bounds on convergence time as a function of the second-largest eigenvalue of the graph. In [10] Bechetti et al. consider a plurality consensus protocol based on a sequence of local majority agreements with three randomly chosen neighbours during each round requiring bias $\Omega(\sqrt{Cn\cdot\log n})$. The protocol converges in $\Theta(\min\{C, n^{1/3}\}\cdot\log n)$ rounds using $\Theta(\log C)$

memory and message size, where C refers to the number of original opinions. In later work [11] the authors solve general plurality consensus in complete graphs via *undecided-state dynamics* using an extra state to accommodate intermediate disagreements. They propose the notion of *monochromatic distance* which reflects on the difference between the initial colour configuration from the closest monochromatic solution. Their plurality protocol converges with a logarithmic overhead on the top of the monochromatic distance. A more recent study on plurality consensus in noisy communication channels can be found in [21].

There is also growing interest in exact-space complexity in probabilistic plurality consensus. In particular, in [12] Berenbrink et al. proposed a plurality consensus protocol converging in $O(\log C \cdot \log\log n)$ synchronous rounds using only $\log C + (\log\log C)$ bits of local memory. They also show a slightly slower solution converging in $O(\log n \cdot \log\log n)$ rounds using only $\log C + 4$ bits of local memory. This disproves a conjecture by Becchetti et al. [11] implying that any protocol with local memory $\log C + O(1)$ has the worst-case running time $\Omega(k)$. In [22] Ghaffari and Parter propose an alternative algorithm converging in $O(\log C \log n)$ rounds while having message and local memory sizes based on $\log C + O(1)$ bits. In addition to the above, some work on the application of the random walk in plurality consenus protocols can be found in [11, 9].

1.2 Our results and organisation of the paper

In this paper we study space-optimal population protocols for several variants of the majority problem. The paper presents space-efficient algorithms for majority with many colours, and these algorithms are obtained by using a combination of known protocols for simple majority. In Section 2 we discus an amendment allowing majority protocols to report a tie (equality) if neither of the two original colours dominates the other. In Section 3 we discuss a solution to the dynamic version of the majority problem in which the original colours assigned to the entities can be changed by an external force. Such a solution is a special case of self-stabilizing population protocols which were considered in [8]. We discuss it here to prepare the ground for our space-optimal protocols for many colours.

We consider space-efficient majority protocols in populations with an arbitrary number C of colours represented by k-bit labels, where $k = \lceil \log C \rceil$. In Section 4 we present an asymptotically space-optimal $O(k)$-bit protocol for the *absolute majority*, i.e., a protocol which answers the question whether one colour dominates all others taken together. In Section 5 we propose a multistage $O(k)$-bit protocol for *relative majority*, where all most frequent colours eventually become aware of their dominance, and all nodes learn about the most frequent colour with the largest label. In Section 6 we conclude with final comments and leave the reader with a list of open problems.

2 Population protocol for static majority with equality

This section reformulates the algorithm for majority presented in [5].

Initially each entity $a \in A$ obtains its original colour c_a, being one of the three available denoted by integers $-1, 0$, and 1. Thus, the main goal in our reformulation of majority protocols is to determine whether there are more 1's than (-1)'s (green domination), more (-1)'s than 1's (red domination), or whether there is a tie between the two. In other words, our majority protocols aim at determining the sign of the expression:

$$\sum_{a \in A} c_a.$$

state s	weight $w(s)$
$[-1]$	-1
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	0
$[1]$	1

Figure 1 The states and their weights.

$s_a \backslash s_b$	$[-1]$	$[0]$	$[1]$	$\langle -1\rangle$	$\langle 0\rangle$	$\langle 1\rangle$
$[-1]$	$([-1],[-1])$	$([-1],\langle -1\rangle)$	$([0],[0])$	$([-1],\langle -1\rangle)$	$([-1],\langle -1\rangle)$	$([-1],\langle -1\rangle)$
$[0]$	$(\langle -1\rangle,[-1])$	$([0],[0])$	$(\langle 1\rangle,[1])$	$([0],\langle 0\rangle)$	$([0],\langle 0\rangle)$	$([0],\langle 0\rangle)$
$[1]$	$([0],[0])$	$([1],\langle 1\rangle)$	$([1],[1])$	$([1],\langle 1\rangle)$	$([1],\langle 1\rangle)$	$([1],\langle 1\rangle)$
$\langle -1\rangle$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle -1\rangle,\langle -1\rangle)$	$(\langle -1\rangle,\langle 0\rangle)$	$(\langle -1\rangle,\langle 1\rangle)$
$\langle 0\rangle$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle 0\rangle,\langle -1\rangle)$	$(\langle 0\rangle,\langle 0\rangle)$	$(\langle 0\rangle,\langle 1\rangle)$
$\langle 1\rangle$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle 1\rangle,\langle -1\rangle)$	$(\langle 1\rangle,\langle 0\rangle)$	$(\langle 1\rangle,\langle 1\rangle)$

Figure 2 The transition table for static majority protocol with ties.

If this sign is positive, there are more 1's, if negative, there are more (-1)'s, and if the sum is 0, we report equivalence between the two competing colours. During the communication process each entity $a \in A$ has an attributed *state* s_a. In due course we will also use the notion of *knowledge* of entities, which includes information about the state and the original colour of the entity.

Throughout the computation process the entities can be in one of the three *strong states* $[-1], [0]$, and $[1]$ or the three weak states $\langle -1\rangle, \langle 0\rangle, \langle 1\rangle$. In the begining, each entity $a \in A$ with attributed colour $c_a = x$ is in state $[x]$. With each state s we associate a weight $w(s)$ such that $w([x]) = x$ and $w(\langle x\rangle) = 0$. This association is illustrated by the table in Fig. 1.

In due course, when two entities $a, b \in A$ interact the shared *transition function* determines their resulting states. And, in particular, if an entity in a strong state $[x]$ meets another in a weak state $\langle y\rangle$, the weak state becomes $\langle x\rangle$ and the strong state remains unchanged. If during a meeting a strong state $[x]$, for $x \neq 0$, meets $[0]$ then only state $[0]$ is changed to $\langle x\rangle$. Finally, if $[1]$ interacts with $[-1]$ both states are changed to $[0]$. Other type of encounters does not change the states of entities. The respective shared transition function is illustrated by the table in Fig. 2.

▶ **Lemma 1** (Invariant 1). *Initially, the sum* $S = \sum_{a\in A} w(s_a)$ *equals to* $\sum_{a\in A} c_a$*, and its value remains unchanged during the computation process.*

Proof. Follows directly from the definition of the transition function. ◀

Observation If the sum S is negative, it declares majority of reds (denoted by -1), positive S indicates majority of greens (denoted by 1), otherwise S refers to the tie.

▶ **Lemma 2** (Invariant 2). *The value of the sum* $R = \sum_{a\in A} |w(s_a)|$ *decreases monotonically throughout the communication process and it stabilises eventually on the value* $R_{\text{fin}} = |S|$.

Proof. At any stage of the algorithm R represents the number of strong states $[-1]$ and $[1]$ still present in the population. According to the transition function the number of such states can only decrease when two states $[1]$ and $[-1]$ annihilate one another during a direct interaction. Thus, eventually the sum R stabilises on the original difference between the number of strong states $|S|$. ◀

We conclude this section with a theorem.

▶ **Theorem 3.** *The population protocol presented in this section computes majority and returns equality if neither of the colours dominates the other.*

Proof. According to the observation and the two lemmas, if a majority exists, the remaining entities in strong states of the dominating colour will recolour all entities accordingly. Otherwise, the annihilation of the last pair of states $([1], [-1])$ results in obtaining two entities with states $[0]$ which in due course will change states in all other entities to $\langle 0 \rangle$. Finally, if neither of the states $[1]$ or $[-1]$ is initially present in the population all entities remain in the neutral state $[0]$. ◀

3 Population protocol for dynamic majority with equality

In this section we consider a variant of population protocols in which the original colours (attributes) of entities could be altered by an *external force* for some unspecified, however limited, period of time. After this initial period, the relevant population protocol is expected to eventually stabilize. The model of changing inputs from [4] and the concept of composing several population protocols as described in [24] are the inspirations for our approach here. In essence, we reformulate the majority algorithm from [4] and show how to modify this reformulation so that it can be used as a subprotocol for our next section.

We assume that an entity is aware when its original colour changes, and is able to modify its current state as a result, but such a change is again governed by common state transition rules for all entities. We also assume that it is not possible for the external force to alter the original colour of an entity while it is simultaneously interacting with another entity.

We use the protocol we propose here as a subroutine in more structurally complex population protocols for the absolute majority in Section 4, and for the relative majority in Section 5.

The population protocol presented below determines whether there are more original 1's, more (-1)'s, or there is a tie after the last intervention of the external force. For the purpose of our protocol each entity $a \in A$ must store its original colour $c_a \in \{-1, 0, 1\}$, and this stored colour can be altered only by the external force at any time. Besides the colour, the entity maintains a state s_a governed by the shared transition function. More formally, an entity's knowledge refers to the pair (c_a, s_a). We define five strong states:$[-2], [-1], [0], [1], [2]$, and three weak states $\langle -1 \rangle, \langle 0 \rangle, \langle 1 \rangle$. Before the protocol is initiated, if $c_a = x$ we set $s_a = [x]$. On the conclusion all entities are in state

- $[1], [2]$ or $\langle 1 \rangle$ if there are more 1's than (-1)'s,
- $[-1], [-2]$ or $\langle -1 \rangle$ if there are less 1's than (-1)'s, and
- $[0]$ or $\langle 0 \rangle$ when there is a tie.

We define the weight function, $w(s)$, on a state s as $w([x]) = x$ and $w(\langle x \rangle) = 0$, see the table in Fig. 3.

During execution of the majority protocol we maintain two invariants:

1. $\sum_{a \in A} c_a = \sum_{a \in A} w(s_a)$, and
2. for each $a \in A$, $|w(s_a) - c_a| \leq 1$.

The two invariants are preserved thanks to carefully crafted state transition rules and counterparting alterations of an entity's state caused by changes of the original colour c_a imposed by the external force. When the colour c_a is changed to $c'_a = c_a + \delta$, the state is changed from s_a to $s'_a = [w(s_a) + \delta]$. Note that this rule preserves both invariants 1 and 2. This is illustrated by the table to the right in Fig. 3 describing how states are changed when

s	$w(s)$
$[-2]$	-2
$[-1]$	-1
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	0
$[1]$	1
$[2]$	2

s_a, $c_a = 1$ changes to $c'_a = -1$	s'_a
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	$[-2]$
$[1]$	$[-1]$
$[2]$	$[0]$
s_a, $c_a = -1$ changes to $c'_a = 1$	s'_a
$[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle$	$[2]$
$[-1]$	$[1]$
$[-2]$	$[0]$

Figure 3 The weight function $w(s)$ and the state transition rules when recolouring occurs by an external force.

$s_a \backslash s_b$	$[-2]$	$[-1]$	$[0]$	$[1]$	$[2]$
$[-2]$	$([-2],[-2])$	$([-2],[-1])$	$([-2],\langle -1\rangle)$	$([-1],\langle -1\rangle)$	$([0],[0])$
$[-1]$	$([-1],[-2])$	$([-1],[-1])$	$([-1],\langle -1\rangle)$	$([0],[0])$	$(\langle 1\rangle,[1])$
$[0]$	$(\langle -1\rangle,[-2])$	$(\langle -1\rangle,[-1])$	$([0],[0])$	$(\langle 1\rangle,[1])$	$(\langle 1\rangle,[2])$
$[1]$	$(\langle -1\rangle,[-1])$	$([0],[0])$	$(\langle 1\rangle,[1])$	$([1],[1])$	$([1],[2])$
$[2]$	$([0],[0])$	$([1],\langle 1\rangle)$	$([2],\langle 1\rangle)$	$([2],[1])$	$([2],[2])$
weak	$(\langle -1\rangle,[-2])$	$(\langle -1\rangle,[-1])$	$(\langle 0\rangle,[0])$	$(\langle 1\rangle,[1])$	$(\langle 1\rangle,[2])$

Figure 4 The state transition table for interacting entities for dynamic majority.

$c_a = 1$ is changed to $c'_a = -1$, or vice-versa. In this table we do not consider, for example, combinations of states $s_a = [-1], [-2]$ with colour $c_a = 1$ because of the invariant 2.

In what follows we describe what happens to the states when two entities $a, b \in A$ interact. If a strong state $[1]$ or $[2]$ meets a weak state $\langle y\rangle$ or $[0]$, then this second state becomes $\langle 1\rangle$. If a strong state $[-1]$ or $[-2]$ meets a weak state $\langle y\rangle$ or $[0]$, then the latter state becomes $\langle -1\rangle$. If a strong state $[0]$ meets a weak state, the weak state is changed to $\langle 0\rangle$. If $[1]$ meets $[-1]$ or $[2]$ meets $[-2]$, they are both changed to $[0]$. If $[2]$ meets $[-1]$, they are changed to $[1]$ and $[0]$ respectively. If $[-2]$ meets $[1]$, they are changed to $[-1]$ and $[0]$ respectively. Other encounters do not result in state alteration. This is illustrated by the table in Fig. 4 which does not take into account encounters between entities where both are in weak states, because they do not result in state alteration.

▶ **Lemma 4.** *The invariants 1 and 2 are preserved during execution of the majority protocol.*

Proof. First, we consider interactions between pairs of entities.

Invariant 1 is preserved, because for any state transition, if the weight of one entity is reduced, then the weight of the other is increased by the same (absolute) value. Also, if colour c_a is changed, then the weight $w(s_a)$ is changed too by the same value.

Invariant 2 is preserved because during every interaction of entities $|w(s_a)|$ can only decrease and $w(s_a)$ does not change its sign. So if $c_a = 1$, then s_a is initially in the interval $[0, 2]$ and it remains in this interval. The reasoning in the remaining cases when $c_a = 0$ or -1 is analogous.

Now we consider the invariants when the external force changes the colour of an entity. Suppose that an entity is coloured $c_a = 1$ and its colour is changed to $c'_a = -1$ (the other case will be similar).

Invariant 1 is preserved by the choice of the transitions shown in the table in the right of Fig. 3. The left hand side of the equation in invariant 1 decreases by 2 (since the colour changes from 1 to -1). If the state of the entity was $s_a \in \{[0], \langle -1\rangle, \langle 0\rangle, \langle 1\rangle\}$, the new state

is $s'_a = [w(s_a) - 2] = [-2]$. Hence the corresponding weight changes from $w(s_a) = 0$ to $w(s'_a) = -2$, so the right hand side of invariant 1 also decreases by 2 (i.e., preserving the invariant). Similarly, if $s_a = [1]$, then the new state is $s'_a = [-1]$, hence the contribution to the right hand side of invariant 1 from the entity changes from $w(s_a) = 1$ to $w(s'_a) = -1$, again a decrease by 2. We can check the remaining case, where $s_a = [2]$, in an analogous manner.

Invariant 2 is also maintained by the rules that govern how the entity's state is updated when its colour is changed by an external force. E.g., $c_a = 1$ changing to $c'_a = -1$ means that the new weight $w(s'_a) \in \{0, -1, -2\}$ from the rules in Fig. 3. ◀

▶ **Lemma 5.** *The value of $R = \sum_a |w(s_a)|$ does not increase after the last intervention of the external force. Moreover the value of R stabilises when eventually there are no two entities $a, b \in A$ such that $w(s_a) > 0$ and $w(s_b) < 0$.*

Due to Lemma 5 the majority process stabilises in three possibile configurations with respect to $C_{\text{fin}} \stackrel{def}{=} \sum_{a \in A} c_a$ (where c_a is referring to the *final* colour of the entity a, after any external forces have stopped changing the colours of entities). If on the conclusion $C_{\text{fin}} > 0$, there must be some entities in states $[1]$ or $[2]$ which would earlier ensure that all weak states and the state $[0]$ are switched to $\langle 1 \rangle$. If $C_{\text{fin}} < 0$, there must be some entities in states $[-1]$ or $[-2]$ which would earlier ensure that all weak states and the state $[0]$ are switched to $\langle -1 \rangle$. However, if on the conclusion $C_{\text{fin}} = 0$, there are no entities in states $[x]$ with $x \neq 0$ and the last entity that reached state $[0]$ will have a chance to alter all weak states to $\langle 0 \rangle$.

4 Absolute majority

In the remaining part of the paper we work under the assumption that the population is coloured with an arbitrary number C of colours, where $2^{k-1} < C \leq 2^k$, for some integer $k \geq 1$ that is known to all entities. Each colour is denoted by a k-bit label $l[0..k-1]$, and single labels are attributed to entities with the relevant colours. As in previous sections, we interpret the individual bits $l[i]$ in this label as -1 or 1, rather than more standard 0 or 1. Each entity is assumed to own an extra $O(k)$ bits used to support the computation process, including interaction with other entities in the population.

In this section we present an asymptotically optimal $O(k)$-bit population protocol computing *absolute majority*, i.e., answering whether there exists a colour which dominates all the remaining colours in the population taken together. The absolute majority algorithm presented here is a combination of the static majority protocol introduced in Section 2, and later referred to as P_1, as well as the dynamic majority protocol from Section 3, from now on referred to as P_2. We recall that protocol P_2 assumes full knowledge of entities and it is using two types of state transitions: (1) imposed by the *external force* and altering original colours associated with entities, and (2) caused by the interaction with other entities in the population.

Memory organisation. Each entity uses $O(k)$ bits of memory to accommodate:

1. The k-bit label $l[0..k-1]$ representing the original colour of the entity,
2. An array $s[0..k-1]$ representing k independent instances of protocol P_1, and
3. An instance of protocol P_2 with the *external force* based on k instances of P_1.

For the purpose of our algorithm we define k independent instances of static majority protocols $P_1(i)$, for $i = 0, \ldots, k-1$, such that colours competing in $P_1(i)$ refer to the bits $l[i]$ drawn from each entity in the population. Assume $l^*[0..k-1]$ is a k-bit label of the colour of

the absolute majority in the population. One can observe that when the majority protocols stabilise, for all $i = 0, \ldots, k-1$, each bit $l^*[i]$ must be in majority reported by $P_1(i)$ via entry $s[i]$. Thus, if the absolute majority exists, one can run k static majority protocols $P_1(i)$ to determine the majority colour. However, if there is no absolute majority the protocol proposed above may still return a false positive "winner". This can happen, e.g., if no entity has a colour with the label in which all bits are set to 1s but the majority of bits $l[i]$, for all $i = 0, .., k-1$ for all entities are 1's. In such case, the non-existing colour with the label filled with 1s would be wrongly recognised by the entities as the absolute majority. In order to overcome this clear deficiency of the protocol, an extra (final) test is performed with the help of protocol P_2 to decide whether the returned colour is in the absolute majority.

4.1 Algorithm Absolute-Majority

Initialisation Stage

1. Before execution of the algorithm, each entity $a \in A$ sets for itself $s[i] = [1]$ if $l[i] = 1$ and $[-1]$ otherwise, for all $i = 0, .., k-1$. This choice refers to the belief that its original colour c_a is in majority. And, indeed, each entity initially adopts an extra colour 1 (denoting membership in the majority) for the purpose of protocol P_2.
2. Later, during pairwise interactions between entities, the current states in $s[i]$ get updated by the relevant majority protocols $P_1(i)$, for each $i = 0, \ldots, k-1$ independently. And if at any time the contents of $s[i]$ and $l[i]$ do not reflect its initial setting, the belief of the entity changes to -1. However, this belief becomes 1 again as soon as the consistency between bits in $s[0..k-1]$ and $l[0..k-1]$ is restored. This consistency measure determines actions of the *external force* in protocol P_2.

Stabilisation Stage

1. At first, the majority algorithm stabilises on all protocols $P_1(i)$, for $i = 0, \ldots, k-1$, which allows each entity to establish the final relationship between the corresponding bits in $s[0..k-1]$ and $l[0..k-1]$. This, in turn, determines the extra colour (1 or -1) of the entity adopted for the purpose of protocol P_2.
2. When eventually protocol P_2 also terminates and concludes with colour 1 in majority, all entities receive confirmation that the final states in $s[0..k-1]$ refer to the absolute majority colour $l^*[0..k-1]$. Otherwise, the entities learn that none of the colours is in the absolute majority.

Note that all protocols described above run simultaneously right from the beginning, and, in particular, protocol P_2 works at least for some time on unstable data. Nevertheless, as the bits generated by protocols P_1 eventually stabilise, thanks to protocol P_2's tolerance of dynamic changes, the absolute majority (if such exists) is confirmed. We conclude with this theorem.

▶ **Theorem 6.** *Algorithm Absolute-Majority computes absolute majority on populations with at most 2^k colours with the help of $O(k)$ memory bits in each entity.*

Proof. If an absolute majority colour exists (represented as a k-bit label $l[0..k-1]$) then, when the k independent instances of P stabilize, each $P_1(i)$ stabilizes in the bit $l(i)$. In fact, each bit of the label of the colour of the absolute majority is then reported by $P_1(i)$ via its entry $s[i]$. However, the population still needs to verify this since, in case of no absolute majority colour, the above protocol may return a false positive "winner" . This can happen if for each i there is an absolute majority bit but the whole tuple of these

bits does not correspond to a colour in the population. In order for this case not to be wrongly understood as the absolute majority , we need a verifying step. This is exactly what protocol P_2 does. In fact, $P(2)$ always runs a test to decide whether the returned supposed absolute majority colour is indeed the absolute majority. Protocol P_2 works for some time on unstable data. However, after a time t by which all $P_1(i)$ have stabiized, protocol P_2 shall also stabilize either by concluding that the assumed majority colour (indicated by colour 1 in the algorithm) is indeed an absolute majority, or it shall stabilize reporting nonexistence of the absolute majority colour to all entities. Note that each time P_2 has to check only one supposed majority colour against all others, treated as a single colour -1 in the algorithm. The above proof works due to the established fact that P_2 tolerates dynamic changes in the input colours. ◀

5 Relative majority

As in Section 4, in this section we assume that the population is attributed with an arbitrary number C of colours, where $2^{k-1} < C \leq 2^k$, for some integer $k \geq 1$ that is known to all entities. Each colour is denoted by a k-bit label $l[0..k-1]$, where $l[i] \in \{-1, 1\}$. Each entity is assumed to have extra $O(k)$ bits used to support the computation process, including communication with other entities in the population. The *relative majority* problem refers to the task of finding the most frequent colour in the population. Note that there can be more than one colour that is the most frequent. In such case the colour with the latest in the lexicographical order label $l^*[0..k-1]$ is declared as the *winner*.

Computing relative majority is a more complex task, comparing to the absolute majority, as here one needs to collect evidence confirming that the winning colour beats any other colour in the population. At first we describe a protocol for the relative majority which only finds the winner $l^*[0..k-1]$. This is done by marking all entities possessing this colour with the winning label. In this setting, the colour in the relative majority always exists. The case in which the uniqueness of the majority colour is required is commented later in Section 5.2.

In the relative majority protocol, instead of engaging in the total comparison (via majority computation) in pairs formed of any two colours, which would require $O(k^2)$-bit memories, we propose a solution similar to finding maximal elements in parallel stages based on duels. In each stage the winning colours perform pairwise duels via majority protocols to reduce the number of winners by half. This multi-stage computation is made feasible thanks to pipelining of dynamic majority protocols P_2 which gradually stabilise starting from the lowest stage and finishing at the highest stage of the dueling process.

Stages are enumerated by descending numbers from the lowest stage $k-1$ to the highest 0. In stage i, for all $i = k-1, .., 0$, two colours are in the same group if their k-bit labels $l[0..k-1]$ share i-bit prefix $l[0..i-1]$ (in stage 0 all labels form one group). In this stage agents in one group aim at finding the majority colour label in each group.

Memory organisation. Each entity $a \in A$ uses $O(k)$ bits of memory to accommodate:

1. The k-bit label $l[0..k-1]$ representing the original colour of the entity. This colour is fixed (never changed) throughout the computation process.
2. The k-bit label $c[0..k-1]$ represents current colours $c[i]$ of the entity in each consecutive stage i, with the decreasing index $i = k-1, .., 0$. On the conclusion of stage i, if label $l[0..k-1]$ is declared as the winner in the group of labels with prefix $l[0..i]$, the value $c[i]$ equals to ± 1, otherwise $c[i] = 0$. All entities with the winning colour $l[0..k-1]$ in

its group in higher stage $i-1$ have the value $c[i]$ set to $l[i]$. Before the stabilisation of $P_2(i-1)$ the value $c[i]$ reflects the current belief of the entity about this value.

3. An array $s[0..k-1]$ representing states $s[i]$ in k independent instances of protocol $P_2(i)$ associated with colours $c[i]$. The computations with respect to $P_2(i)$ are performed only if the two interacting entities have the same label prefix $l[0..i-1]$. Otherwise protocol $P_2(i)$ is not executed. We emphasise here that computations in $P_2(i)$ can change values $c[i-1]$ whose change in turn cause alteration of states $s[i-1]$. Also, changes in $c[i]$ can change $c[i-1]$.

5.1 Algorithm Relative-Majority

Initialisation Stage

Before execution of the algorithm, each entity sets $c[i] = l[i]$ and $s[i] = [1]$ if $c[i] = 1$ and $[-1]$ otherwise, for all $i = 0, .., k-1$.

Stabilisation Stage

1. The algorithm stabilises first on protocol $P_1(k-1)$, as at the beginning of the pipeline there is no external force, and then subsequently on protocols $P_2(k-2)$, $P_2(k-3)$, ..., $P_2(0)$.
2. An entity believes that its colour wins on stage i if, either $c[i] = -1$ and $s[i] \in \{[-1], [-2], \langle -1\rangle\}$, or $c[i] = 1$ and $s[i] \in \{[0], [1], [2], \langle 0\rangle, \langle 1\rangle\}$. The states $[0], \langle 0\rangle$ correspond to a tie and in this case the lexicographically larger label becomes the winner. If the entity believes its label $l[0..k-1]$ is the winner in stage i, it sets $c[i-1] = l[i-1]$ and adjusts $s[i-1]$ as specified in protocol $P_2(i-1)$ if $c[i-1]$ gets changed. If, to the contrary, the entity believes it did not win, it sets $c[i-1] = 0$ and also adjusts $s[i-1]$ should change occur in $c[i-1]$. Note, that in both cases changes in $c[i-1]$ are propagated to $c[i-2]$ and further on.
3. Eventually protocol $P_2(0)$ stabilizes. At that time entities that win in stage 0 hold the winning majority colour.

▶ **Theorem 7.** *Algorithm Relative-Majority computes relative majority on population with at most 2^k colours with the help of $O(k)$ memory bits in each entity.*

Proof. The memory requirement follows directly from the formulation of the protocol. In order to prove correctness, we proceed by induction on stage numbers i taken in reverse order. The colours $c[k-1]$ do not change during the protocol so in some moment t_{k-1} protocols $P_2(k-1)$ stabilize and states $s[k-1]$ stop being changed. These states determine unique winning k-bit colours in groups corresponding to all possible prefixes $l[0..k-2]$.

Now let $i > k-1$ be a stage number. By inductive hypothesis in some time t_{i+1}, protocols $P_2(i+1)$ stabilize and states $s[i+1]$ stop being changed. They indicate unique winning k-bit colours in groups corresponding to each prefix $l[0..i]$. So, since t_{i+1} colours $c[i]$ are ± 1 for these winners, 0 for others and do not change anymore. Thus, in some later time t_i, protocols $P_2(i)$ stabilize and states $s[i]$ cease being changed. From the formulation of the protocol these final states $s[i]$ determine the winning k-bit colours in groups corresponding to prefixes $l[0..i-1]$.

Finally ,at some time t_0, protocols $P_2(0)$ stabilize and all entities compute states $s[0]$ corresponding to the unique winning k-bit colour amongst all of them. ◀

5.2 Uniqueness in relative majority

As indicated at the beginning of Section 5, one may want to report only unique relative majority colours, i.e., when there is exactly one, the most frequent colour. And indeed if the winning colour l^* is not unique, there must exist some other colours which lost to l^* in a tie at some stage. The purpose of the mechanism presented below is to encounter such ties (if they exist) and to distribute this information to all entities in the population. This can be done by performing an additional *dissemination protocol* with the help of an extra bit c' drawn from the set $\{0, 1\}$. This dissemination protocol is run by each entity in conjunction with the relative majority protocol described above, and its actions are governed by the current belief of the entity whether it is a winner or not and by encountered or not ties in duels. The following four rules govern values of the extra bit c'.

Initially, (1) in each entity the extra bit c' is set to 0 to denote that the entity does not carry any information about ties between the winners. This value can be changed to 1 if (2) the colour of the entity is still a potential winner (did not lose any duel yet in the most recent climb through the stages) and at some stage its duel ends up in a tie; or if (3) the colour of the entity is already deemed as the loser and it meets another entity with the colour still being a potential winner and its extra bit $c' = 1$. And (4) the extra bit c' can be changed back to 0 if and only if the colour of its owner is deemed as loser and it meets another entity with the colour still being a potential winner and its extra bit $c' = 0$.

In due course the values of each extra bits c' can be altered several times according to the rules 1, 2 or 3. However, when eventually the relative majority protocol determines the winning colour l^* in stage 0, only entities coloured with l^* are able to change values of extra bits in other entities. Now, if the extra bit associated with entities coloured by l^* is 0, i.e., the winning colour has never experienced a tie, all other entities are eventually informed accordingly by rule 4. And, if the extra bit associated with entities coloured by l^* is 1, i.e., the winning colour has encountered a tie in the past, all other entities are eventually informed accordingly by rule 3.

6 Conclusion

In this paper we presented memory-efficient population protocols for several variants of the majority problem.

In Section 2 we show how to amend majority protocols to report ties. The proposed protocol relies on a relatively large number of states used by entities. One can show a more space-efficient solution limited to six states. Also in a wider context, in our solutions the emphasis was on asymptotic space optimality. One open problem, however, is to determine more exact bounds on the number of states required to compute the considered types of majorities for a given number of colours C. Another interesting problem refers to the time complexity and parallelism of considered majority problems in the presence of a random scheduler. Finally, one can ask what other computations are possible through a composition of several "partially self-stabilizing" (sub)protools.

Acknowledgements. The authors wish to thank Yukiko Yamauchi for the valuable referral to [18] about fair composition of self-stabilising algorithms, and for discussions related to the work in this paper.

We also wish to thank the anonymous referees and the shepherd of OPODIS 2016 for their help in preparation of the final version of this paper.

References

1. D. Alistarh, J. Aspnes, D. Eisenstat, R. Gelashvili, and R. L. Rivest. Time-space trade-offs in population protocols. CoRR abs/1602.08032, 2016.
2. D. Alistarh and R. Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 479–491, 2015.
3. D. Alistarh, R. Gelashvili, and M. Vojnovic. Fast and exact majority in population protocols. In *Proc. 33rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 47–56, 2015.
4. D. Angluin, J. Aspnes, M. Chan, M.J. Fischer, H. Jiang, and R. Peralta. Stably computable properties of network graphs. In *Proc. 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 63–74, 2005.
5. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. In *Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 290–299, 2004.
6. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
7. D. Angluin, J. Aspnes, and D. Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.
8. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3(4):1–28, 2008.
9. L. Gąsieniec, D. D. Hamilton, R. Martin, and P. G. Spirakis. The match-maker: Constant-space distributed majority via random walks. In *Proc. 17th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 67–80, 2015.
10. L. Becchetti, A. Clementi, E. Natale, F. Pasquale, R. Silvestri, and L. Trevisan. Simple dynamics for plurality consensus. In *Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 247–256, 2014.
11. L. Becchetti, A. E. F. Clementi, E. Natale, F. Pasquale, and R. Silvestri. Plurality consensus in the gossip model. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 371–390, 2015.
12. P. Berenbrink, T. Friedetzky, G. Giakkoupis, and P. Kling. Efficient plurality consensus, or: The benefits of cleaning up from time to time. In *Proc. 43rd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 1–14, 2016.
13. J. M. Bower and H. Bolouri. *Computational modeling of genetic and biochemical networks.* MIT Press, 2004.
14. H.-L. Chen, R. Cummings, D. Doty, and D. Soloveichik. Speed faults in computation by chemical reaction networks. *Distributed Computing*, pages 16–30, 2014.
15. C. Cooper, R. Elsaesser, and T. Radzik. The power of two choices in distributed voting. In *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 435–446, 2014.
16. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert. When birds die: Making population protocols fault-tolerant. In *IEEE 2nd Intl. Conference on Distributed Computing in Sensor Systems (DCOSS)*, volume 4026 of *Lecture Notes in Computer Science*, pages 51–56. Springer-Verlag, 2006.
17. B. Doerr, L. A. Goldberg, L. Minder, T. Sauerwald, , and C. Scheideler. Stabilizing consensus with the power of two choices. In *Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 149–158, 2011.
18. S. Dolev. *Self Stabilization.* MIT Press, 2000.

19 D. Doty and D. Soloveichik. Stable leader election in population protocols requires linear time. In *Proc. 29th International Symposium on Distributed Computing (DISC)*, pages 602–616, 2015.

20 M. Draief and M. Vojnovic. Convergence speed of binary interval consensus. *SIAM Journal on Control and Optimization*, 50(3):1087–1109, 2012.

21 P. Fraigniaud and E. Natale. Noisy rumor spreading and plurality consensus. In *Proc. 34th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 127–136, 2016.

22 M. Ghaffari and M. Parter. A polylogarithmic gossip algorithm for plurality consensus. In *Proc. 34th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 117–126, 2016.

23 G. B. Mertzios, S. E. Nikoletseas, C. Raptopoulos, and P. G. Spirakis. Determining majority in networks with local interactions and very small local memory. In *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 871–882, 2014.

24 O. Michail, I. Chatzigiannakis, and P. G. Spirakis. *New Models for Population Protocols*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

25 O. Michail and P. G. Spirakis. Simple and efficient local codes for distributed stable network construction. In *Proc. 32nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 76–85, 2014.

26 Y. Mocquard, E. Anceaume, J. Aspnes, Y. Busnel, and B. Sericola. Counting with population protocols. In *Proc. 2015 IEEE 14th International Symposium on Network Computing and Applications (NCA)*, pages 35–42, 2015.

27 E. Perron, D. Vasudevan, and M. Vojnovic. Using three states for binary consensus on complete graphs. In *Proc. 28th Conference on Computer Communications (INFOCOM)*, pages 2527–2535, 2009.

Design Patterns in Beeping Algorithms*†

Arnaud Casteigts[1], Yves Métivier[2], John Michael Robson[3], and Akka Zemmari[4]

1 **LaBRI, University of Bordeaux, Bordeaux, France**
`acasteig@labri.fr`
2 **LaBRI, University of Bordeaux, Bordeaux, France**
`metivier@labri.fr`
3 **LaBRI, University of Bordeaux, Bordeaux, France**
`robson@labri.fr`
4 **LaBRI, University of Bordeaux, Bordeaux, France**
`zemmari@labri.fr`

Abstract

We consider networks of processes which interact with beeps. In the basic model defined by Cornejo and Kuhn [5], which we refer to as the BL variant, processes can choose in each round either to beep or to listen. Those who beep are unable to detect simultaneous beeps. Those who listen can only distinguish between silence and the presence of at least one beep. Stronger variants exist where the nodes can also detect collision while they are beeping ($B_{cd}L$) or listening (BL_{cd}), or both ($B_{cd}L_{cd}$). Beeping models are weak in essence and even simple tasks are difficult or unfeasible with them.

This paper starts with a discussion on generic building blocks (*design patterns*) which seem to occur frequently in the design of beeping algorithms. They include *multi-slot phases*: the fact of dividing the main loop into a number of specialised slots; *exclusive beeps*: having a single node beep at a time in a neighbourhood (within one or two hops); *adaptive probability*: increasing or decreasing the probability of beeping to produce more exclusive beeps; *internal* (resp. *peripheral*) collision detection: for detecting collision while beeping (resp. listening); and *emulation* of collision detection: for enabling this feature when it is not available as a primitive.

We then provide algorithms for a number of basic problems, including colouring, 2-hop colouring, degree computation, 2-hop MIS, and collision detection (in BL). Using the patterns, we formulate these algorithms in a rather concise and elegant way. Their analyses (in the full version) are more technical, e.g. one of them relies on a Martingale technique with non-independent variables; another improves that of the MIS algorithm in [8] by getting rid of a gigantic constant (the asymptotic order was already optimal).

Finally, we study the relative power of several variants of beeping models. In particular, we explain how *every* Las Vegas algorithm with collision detection can be converted, through emulation, into a Monte Carlo algorithm without, at the cost of a logarithmic slowdown. We prove that this slowdown is optimal up to a constant factor by giving a matching lower bound.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Beeping models, Design patterns, Collision detection, Colouring, 2-hop colouring, Degree computation, Emulation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.15

* Full version available on arXiv (`http://arxiv.org/abs/1607.02951`).

† This research has been supported by ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16-CE25-0009-03).

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 15; pp. 15:1–15:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Distributed computing is concerned with various assumptions, like the structure of the network (trees, rings, planar graphs, *etc.*) or knowledge available to the nodes (network size, identifiers, port numbering, *etc.*). Another important aspect is the size of messages, which may range from unbounded, to logarithmic size, to constant size.

As a natural goal is to reduce assumptions as much as possible. Typically, when a problem is solved in some strong model, the community strives to solve it in weaker models. In a recent series of works [5, 10, 1, 7, 8, 6], new models were explored that are even weaker than constant size messages. They are called *beeping models*.

In beeping models, the only communication capabilities offered to the nodes are to *beep* or to *listen*. Several variants exist. In [5], a node that beeps is unable to detect whether other nodes have beeped simultaneously. When listening, it can distinguish between silence or the presence of at least one beep, but it cannot distinguish between one and several beeps. In Section 6 of [1], beeping nodes can detect whether other nodes are beeping simultaneously. In [10] and Section 4 of [1], yet another variant is considered where listening nodes can tell the difference between silence, one beep, and several beeps.

In this paper, we denote the ability to detect collision while beeping (internal collision) by B_{cd} and that of detecting collision while listening (peripheral collision) by L_{cd}. The absence of such ability is denoted by B and L, respectively. The existing models can be reformulated using the cartesian product of these capabilities. Hence, the basic model introduced by Cornejo and Kuhn in [5] is BL; the model considered by Afek et al. in [1] (Section 6) and Jeavons et al. in [8] is $B_{cd}L$; and the model considered in [10] and in Section 4 of [1] is BL_{cd}. To the best of our knowledge, $B_{cd}L_{cd}$ was only used in a previous work of the authors [3].

Although some variants are stronger than others, all beeping models remain extremely weak in essence. Yet, they are relevant to account for real-world applications or phenomena. For instance, they reflect the features of a network at the lowest levels (physical and MAC layers), where a node can probe or emit signals, with or without collision detection. At a higher level of abstraction, beeping models also reflect some communication patterns in biology [4, 1, 9].

1.1 Contributions

The contributions of this paper are manifold. As a warm-up, we start by identifying generic building blocks (*design patterns*) which seem to occur often in the design of beeping algorithms. Then we present a number of algorithms for various graph problems which improve upon previous solutions. Finally, we generalise existing emulation techniques for using collision detection if it is not available, and we prove them optimal *w.h.p.* up to a constant factor.

Due to space limitations, this version of the paper omits (in its core) most complexity analyses and some proofs. However, both are available in the arXiv version whose reference is given in the first page.

1.1.1 Design patterns

We identify a number of common building blocks in beeping algorithms, including *multi-slot phases*: the fact of dividing the main loop into a (typically constant) number of slots having specific roles (*e.g.,* contention among neighbours, collision detection, termination detection); *exclusive beeps*: the fact of having a single node beep at a time in a neighbourhood (within one or two hops, depending on the needs); *adaptive probability*: increasing or decreasing

Table 1 Randomised Las Vegas colouring algorithms on graphs with n vertices.

Model	Time (# slots)	Message size	Knowledge	# colours
$B_{cd}L$	$O(\log n + \Delta)$ expected and *w.h.p.*	$\simeq 1$ bit ($B_{cd}L$ beeps)	None	$O(\log n + \Delta)$
$B_{cd}L$	$O\left(K(\log n + \log^2 K)\right)$ *w.h.p.*	$\simeq 1$ bit ($B_{cd}L$ beeps)	Upper bound K on the max degree of G	K

the probability of beeping in order to maximise the number of exclusive beeps; *internal* (resp. *peripheral*) collision detection: the fact of detecting collision while beeping (resp. listening); and *emulation* of collision detection: the fact of detecting collisions even when it is not available as a primitive. As we show in the paper, these patterns make it possible to formulate the algorithms in a rather concise and elegant way.

1.1.2 Algorithms and analyses for basic graph problems

We present, or analyse algorithms for a number of basic graph problems, including colouring, 2-hop colouring, degree computation, Maximal Independent Set (MIS) and 2-hop MIS. Quite often, the design of algorithms is easier and more natural if collision detection is assumed as a primitive, e.g., in $B_{cd}L_{cd}$ or $B_{cd}L$. Furthermore, emulation techniques such as those described later in this paper enable safe and automatic translations of algorithms into weaker models like BL. For this reason, our algorithms are expressed using whichever model is the most convenient.

First, we present a Las Vegas (i.e. guaranteed result, uncertain time) colouring algorithm in the $B_{cd}L$ model, with time complexity of $O(\log n + \Delta)$ slots *w.h.p.*, where Δ is the maximum degree in G. Its analysis relies on a martingale technique with non-independent random variables, which makes use of a result by Azuma [2] (details in the long version). In fact, the phenomenon is quite ubiquitous in beeping models: the algorithm terminates in the first moment when every node has produced an exclusive beep at least once within its (1-hop) neighbourhood. This stopping time is made more complex by the use of the *adaptive probability* pattern mentioned above. Another algorithm for 2-hop colouring is given, this time in the $B_{cd}L_{cd}$ model, with slot complexity $O(\log n + \Delta^2)$ *w.h.p.* Both algorithms require no knowledge on G. However, both can result in arbitrarily many colours (in fact, one per slot). If the nodes know an upper bound $K \geq \Delta$, a different strategy is proposed that uses at most $K + 1$ colours. However, the slot complexity becomes $O(K(\log n + \log^2 K))$ *w.h.p.* for colouring (trade K for K^2 in the 2-hop variant). Note that this complexity is not thought to be tight. The results are summarised on Table 1.

Based on the observation that degree computation is strongly related to 2-hop colouring, we present an adaptation of the algorithm for this problem, with same slot complexity, that is, $O(\log n + \Delta^2)$ *w.h.p.* In fact, the random process induced by this algorithm is the same as that of colouring, except that it occurs in the *square* of the graph (whence the Δ^2 term). Algorithmically, the main loop contains more specialised slots (e.g., one for peripheral collision reporting), but still a constant number of them, which keeps the asymptotics unchanged. We then turn our attention to the 2-hop MIS problem, which shares common traits and patterns with 2-hop colouring and degree computation and, regarding the high-level purpose of each phase, with the MIS algorithm from [8]. The running time is however shorter than that of 2-hop coulouring and degree computation (and the analysis quite different) due to the fact that exclusive beeps cause whole neighbourhoods to terminate at once. In fact, we prove that the slot complexity of this algorithm is $O(\log n)$ *w.h.p.* with a "reasonable" constant

factor of 76. Noteworthily, the number of phases (i.e. iterations of the main loop) for the 2-hop MIS is exactly the same as what the analogue for classical MIS would produce in the square of the graph. As a consequence, our analysis also improves substantially that of the MIS algorithm presented in [8], where a gigantic constant factor (i.e. one larger than e^{25}) is used. An earlier analysis in [11] yielded a better, yet huge constant of 2×10^{11}. Although constant factors are less meaningful in general, the gap in this case is one between practical and unpractical running times. Furthermore, the contribution it not as much in the constant itself than in the analysis techniques that achieve it.

1.1.3 Collision detection and emulation techniques

Classical considerations on symmetry breaking in anonymous beeping networks, see for example [1] (Lemma 4.1), imply that there is no Las Vegas internal collision detection algorithm in the beeping models BL and BL_{cd}. Likewise, there is no Las Vegas peripheral collision detection algorithm in the beeping models BL and $B_{cd}L$. Since collision detection is required to detect exclusive beeps with certainty, and this pattern is central in most beeping algorithms, this implies that a large range of algorithms cannot exist in a Las Vegas version in these models.

We study the cost of detecting collision when it is not available, typically in BL, and present generic techniques to emulate collision detection probabilistically in order to transform Las Vegas algorithms with collision detection into Monte Carlo algorithms (uncertain result, guaranteed time) in BL. These techniques generalise that of Algorithm 3 in [1], where a similar strategy is encapsulated into the algorithm. We show how, given $0 < \epsilon < 1$, any collision in the neighbourhood of a *given* node can be detected in $O(\log(\frac{1}{\epsilon}))$ slots with error at most ϵ, and similarly it can be detected in $O(\log n)$ slots *w.h.p.* Ensuring that this is true for *any* node requires more time. By union bound, it holds that $O(\log(\frac{n}{\epsilon}))$ slots are sufficient with error ϵ and that $O(\log n)$ slots are sufficient *w.h.p.* We prove that this technique is essentially optimal (asymptotically and up to a constant factor) by giving a matching lower bound. Precisely, we prove that some topologies require $\Omega(\log n)$ slots to break symmetries *w.h.p.* Finally, we provide two generic procedures that can be used in an algorithm to emulate collision detection when it is not available (e.g. in BL). These procedures are `EmulateB`$_{cd}$`inBL()`, to detect collision while beeping, and `EmulateL`$_{cd}$`inBL()`, to detect collision while listening. We illustrate their use in the case of the computation of a MIS given in $B_{cd}L$, thus obtaining a Monte Carlo algorithm in BL.

1.2 Organisation of the paper

In Section 2 we present the model and give further definitions. Section 3 introduces design patterns in a tutorial manner. These patterns are then used in Section 4 to describe the various algorithms. Finally, Section 5 presents our contribution on collision detection and emulation techniques.

2 Network Model and Definitions

We consider a wireless network and we follow definitions given in [1] and [5]. The network is anonymous: unique identifiers are not available to distinguish the processes. Possible communications are encoded by a graph $G = (V, E)$ where the nodes V represent processes and the edges E represent pairs of processes that can hear each other. We denote by Δ the maximum degree of G. The neighbourhood of a vertex v, denoted $N(v)$, is the set of vertices

adjacent to v (at distance 1 from v). We define $\overline{N}(v)$ by including v itself in $N(v)$. We also use the set of vertices at distance at most 2 from v called the 2-neighbourhood of v and denoted $N_2(v)$ (or $\overline{N_2}(v)$ if it includes v). Finally, we write $\log n$ for the binary logarithm of n.

Time is divided into discrete synchronised time intervals (rounds) also called *slots* (following the usual terminology in wireless networks). All processes wake up and start computation in the same slot. In each slot, all processors act in parallel and either beep or listen. In addition, processors can perform an unrestricted amount of local computation in-between two slots (in effect, our algorithms require little computation).

▶ Remark. In general, nodes are active or passive. When they are active they beep or listen; in the description of algorithms we say explicitly when a node beeps meaning that a non beeping active node listens.

The time complexity, also called *slot complexity*, is the maximum number of slots needed until every node has terminated. Our algorithms are typically structured into *phases*, each of which corresponds to a small (constant or logarithmic) number of slots. In the algorithm, we specify which one is the current slot by means of a `switch` instruction with as many `case` statements as there are slots in the phase. Phases repeat until some condition holds for termination.

▶ Remark. An algorithm given in a beeping model induces an algorithm in the (synchronous) message passing model. Thus, given a problem, any lower bound on the round complexity in the message passing model also holds for slot complexity in the beeping model.

Distributed Randomised Algorithm

A randomised (or probabilistic) algorithm is an algorithm which makes choices based on given probability distributions. A *distributed* randomised algorithm is a collection of local randomised algorithms (in our case, all identical).

A *Las Vegas* algorithm is a randomised algorithm whose running time is not deterministic, but still finite with probability 1, and that always produces a correct result. A *Monte Carlo* algorithm is a randomised algorithm whose running time is deterministic, but whose result may be incorrect with a certain probability. Put differently, Las Vegas algorithms have uncertain execution time but certain result, and Monte Carlo algorithms have certain execution time but uncertain result. Classical considerations on symmetry breaking in anonymous beeping networks (see for instance Lemma 4.1 in [1]), imply that:

▶ Remark. There is no Las Vegas (and a fortiori no deterministic) algorithm in BL which allows a node to distinguish between an execution where it is isolated and one where it has exactly one neighbour.

From this remark we deduce that there is no Las Vegas counting algorithm in BL, which advocates the use of stronger models. In what follows, we consider whichever model is the most convenient and provide Las Vegas algorithms in these models. We then present canonical emulation techniques to turn any such algorithm into a Monte Carlo one in BL.

3 Design patterns for beeping algorithms

As a warm-up, this section presents a number of design patterns which seem to occur frequently in the design of beeping algorithms. The concept of pattern refers here to reusable solutions to common problems. These patterns are then used to describe algorithms in the other sections.

Algorithm 1: Exclusive beeps (using B_{cd}).

```
repeat
    beep with some probability;
    if I beeped alone then
        do something exclusive;
    ...
until finished;
```

Algorithm 2: Two-hops exclusive beeps (using $B_{cd}L_{cd}$).

```
repeat
    switch slot do
        slot 1 // contending
            beep with some probability;
        slot 2 // detection of peripheral collision
            if several neighbours beeped in slot 1 then
                beep
        after slot 2
            if I beeped alone in slot 1 and no neighbour beeped in slot 2 then
                do something 2-hop exclusive
        ...
until finished;
```

Exclusive beeps

Beeping algorithms operate in synchronous periods called *slots*, which are equivalent to the concept of rounds in message passing models. Most problems in distributed computing require some node v to take exclusive decisions at times (i.e., with respect to vertices of $\overline{N}(v)$ or $\overline{N_2}(v)$), which requires some type of symmetry breaking. In beeping networks, this goal is all the more difficult to achieve that the nodes cannot use identifiers nor even port numbers in their basic exchanges. If we assume that a node that is beeping can detect whether another node beeps simultaneously (B_{cd}), then this feature can be used to take exclusive decision if indeed it beeps alone. We call this an *exclusive beep*. Algorithm 1 illustrates an empty shell of algorithm that relies on repeated attempts to produce exclusive beeps. Most, if not all algorithms rely implicitly on this pattern as a basis.

2-hop exclusive beeps

For some problems like 2-hop colouring, 2-hop MIS, or computation of the degree (all discussed in this paper), the level of mutual exclusion offered by exclusive beeps is not sufficient and the algorithm requires that a node be the only one to beep at distance 2. Assuming collision can also be detected upon listening (L_{cd}), one can design a 2-slots pattern whereby non-beeping neighbours report if they have heard more than one beep. Hence, if a node produced an exclusive beep in the first slot, and none of its neighbours reported a collision in the second, then it knows that it has produced a *2-hop exclusive beep* (see Algorithm 2).

Algorithm 3: Adaptive beeping probability (using $B_{cd}L_{cd}$).

```
Float p ← 1/2  // say
repeat
    beep with probability p;
    if I beeped alone then
        do something exclusive;
    else
        if no one beeped then
            increase p;
        else
            decrease p;
until finished;
```

Multi-slot phases

The example in Algorithm 2 illustrates another common aspect of beeping algorithms, namely *multi-slot phases*. The expressivity of a single beep is rather poor, but several combined slots can achieve elaborate behavior. In Algorithm 2, one slot is devoted to contending and another to peripheral collision detection. The whole compound is then called a *phase*. Another common task is termination detection. In a *termination slot*, all nodes which have not yet performed some action beep. If the slot remain silent, then a form of local termination is detected: nodes are in a terminal state.

Adaptive probability

As far as feasibility and expressivity are concerned, the next design pattern is not crucial. However, it plays a central role in terms of performance. *Adaptive probability* consists in adapting the probability to beep in the next phase depending on the outcome of previous phases. Typically, if a collision occurs, the probability is reduced, and if no one beeps, it is increased. Since the nodes do not know how many neighbours are contending with them, this technique proves useful in optimizing the odds of producing exclusive beeps.

The values given to the probabilities in Algorithm 3 are left unspecified. There are several options. In this paper, we use a doubling/halving pattern, that is, p is increased to $2p$ (up to $1/2$), and it is decreased to $p/2$ (without limit). A similar doubling/halving pattern was used in [11]. One could also increment or decrement the denominator of p as done in [3]. The consequences of choosing one over the other are not discussed here.

Collision detection

Most algorithms in this paper use collision detection as a built-in primitive, referred to as B_{cd} for detection on beeping and L_{cd} for detection on listening. However, this feature is not always available as a primitive. An important question is the transformation of a (high-level) algorithm using B_{cd} or L_{cd} (or both) into one that works in the weakest BL model. This question is the topic of Section 5, in which we study generic mechanisms to achieve this goal. Essentially, each slot that requires collision detection can be replaced with a logarithmic number of slots (in the size of various quantities depending on the desired guarantees) where the ties are broken *w.h.p.* We provide dedicated procedures that generalise the technique used internally to one of the algorithms in [1]. Besides complexity, the price to pay is that

the algorithm becomes Monte Carlo instead of Las Vegas, that is, the result is correct only probabilistically (though possibly *w.h.p.*). We present a matching lower bound showing that these procedures are essentially optimal.

4 Algorithms for basic graph problems

We now present algorithms for a number of problems, including colouring (with or without knowledge on the degree), 2-hop colouring, computation of the degree and 2-hop MIS. These algorithms are based on various combinations of the patterns presented in Section 3. All algorithms are Las Vegas, and they rely on medium to strong primitives ($B_{cd}L$ to $B_{cd}L_{cd}$ models) depending on the needs. The adaptation of these algorithms in the weakest model (BL) is discussed in Section 5. We also recall Jeavons et al.'s Las Vegas algorithm for the MIS [8] problem and discuss its relations with our 2-hop MIS algorithm.

Whenever using the adaptive probability pattern in algorithms, for generality, we stick to the terms *increase* and *decrease* (as opposed to our analyses, in which these actions are instantiated to *doubling* and *halving* the probability).

4.1 Colouring

The colouring problem consists of assigning a colour to every node in the network, such that no two neighbours have the same colour. We first consider the case that no extra information is available to the nodes. Then we consider that (an upper bound on) the maximum degree is known.

Colouring without knowledge

Informally, the algorithm proceeds as follows (see Algorithm 4 for details). Initially, every node is uncoloured (*nil*). In every phase, each node increments a counter. Uncoloured nodes contend with each other to produce an *exclusive beep*, and when one succeeds, it takes the current value of the counter as its colour and retires. An *adaptive probability* is used to regulate the probability of beeping among uncoloured nodes. Local termination (a node and its neighbours are coloured) detection is not explicitly handled here, though we could add a *termination slot* where uncoloured nodes are the only ones to beep.

The running time of this algorithm is of $O(\log n + \Delta)$ phases *w.h.p* as well as on average (none of both imply the other trivially). Note that this is also the number of slots, since each phase consists of a *constant* number of slots. As for the number of colours, it is incremented with time, thus it is at most the same (at most, because some phases may not produce exclusive beeps).

Colouring with a bound K on the maximum degree Δ

If a bound $K \geq \Delta$ is known, then one can obtain a better colouring using at most $K + 1$ colours. The algorithm follows the same lines as Algorithm 4, i.e. a colour counter is incremented in each phase, and its current value is chosen by those nodes who produced an exclusive beep. The main difference (see Algorithm 5 for details) is that only those colours within $\{0, \ldots, K\}$ are considered and thus the counter is incremented modulo $K + 1$. Conflicts of colours are avoided by keeping a phase idle if the corresponding value was already taken in the past (locally). To do so, when a node takes a colour, it *re-beeps* in a new slot called *confirmation slot* to inform its neighbours that they must remove the current colour from their list of authorized colours. Accordingly, the uncoloured will contend in a phase

Algorithm 4: A Las Vegas colouring algorithm in $B_{cd}L$ (without knowledge).

```
Float p ← 1/2;
Integer colour ← nil;
Integer counter ← 0;
repeat
    beep with probability p;
    if I beeped alone then
        colour ← counter
    else
        if no one beeped then
            increase p;
        else
            decrease p;
        counter ← counter + 1;
until colour ≠ nil;
```

only if the current colour is still available (otherwise, they wait). An adaptive probability is used similarly to Algorithm 4, except that idle phases are not considered as silent (the probability is not updated in these phases).

Regarding performance, the only difference between this algorithm and Algorithm 4 is that a growing number of phases are idle in each neighbourhood, inflicting a slow down to the algorithm. Managing the dependencies here proved more difficult and we "only" managed to prove that the number of phases is $O\left(K(\log n + \log^2 K)\right)$ *w.h.p.* However, the algorithm is believed to be faster.

4.2 2-hop colouring

A 2-hop colouring of a graph G is a colouring such that any two nodes at distance ≤ 2 have different colours. In other words, it is a colouring of the square of G, the graph where an edge exists between nodes which are neighbours in G or share a common neighbour in G.

2-hop colouring without knowledge

A similar strategy is used as in Algorithm 4 (colouring), except that exclusive beeps are replaced with *2-hop exclusive beeps*. Whenever a node produces such a beep, it takes the current value of the counter as colour. Since no other node has beeped within distance 2, the colouring is legal. Contrary to the 1-hop colouring, the collaboration of a node remains crucial even after it becomes coloured. Indeed, this node must keep on reporting peripheral collisions to its neighbours. As a result, instead of retiring from computation, coloured nodes keep on listening until all of their neighbours are coloured, which is detected using an extra *termination slot*. Details are given in Algorithm 6. Four slots are used in total, the first two being devoted to the management of 2-hop exclusive beeps (see Section 3 for details). The third slot manages a (2-hop) adaptive probability based on beeps heard at distance one (slot 1) or at distance two (slot 3 itself). Finally, slot 4 is the termination slot.

Once we realize that the execution produced here is the same as what Algorithm 4 would produce in the square of G, analysis of this algorithm is straightforward. The only difference is that the maximal number of contenders of a node becomes Δ^2 instead of Δ. Thus Algorithm 6 takes $O(\log n + \Delta^2)$ phases (and slots) *w.h.p.*, and the number of colours cannot exceed the same value.

Algorithm 5: A Las Vegas colouring algorithm in $B_{cd}L$ (knowing $K \geq \Delta$).

```
Colours = {0, ··· , K};
Float p ← 1/2;
Integer colour ← nil;
Integer counter ← 0;
repeat
    if counter ∈ Colours then
        switch slot do
            slot 1 // contending
                beep with probability p
            slot 2 // confirmation
                if I beeped alone in slot 1 then
                    colour ← counter;
                    beep;
                else
                    if no one beeped then
                        increase p;
                    else
                        decrease p;
                if someone beeped in slot 2 then
                    Colours ← Colours \ {counter}
    counter ← (counter + 1)  mod (K + 1);
until colour ≠ nil;
```

With a bound K on the maximum degree Δ

The same idea can be applied as in the 1-hop variant, *i.e.*, taking colours between 0 and $K^2 + 1$ (instead of $K + 1$) and incrementing the counter accordingly (mod $K^2 + 1$). As a result, at most $K^2 + 1$ colours are used, with time complexity $O(K^2(\log n + \log^2 K))$ *w.h.p.*

4.3 Degree computation

Let us recall that 2-hop exclusive beeps allow a node v to perform an exclusive action within a radius of distance 2. This feature was used in Section 4.2 to assign unique colours. At it turns out, the pattern is very versatile and it can be used to count the degree of a node as well. The strategy consists in replacing the colour-related action in slot 2 (second `if-then` block) by an action aiming at having v counted in the degree of its neighbours (then v stops contending and keeps on reporting collisions, as before). Precisely, a new confirmation slot is inserted wherein v re-beeps if indeed it produced a 2-hop exclusive beep. Upon hearing the confirmation beep, all of v's neighbours increment a local counter that eventually amounts to their degree. Termination proceeds in the same way as for the 2-hop-colouring algorithm (i.e. uncounted nodes beep in a termination slot).

Up to a constant factor which accounts for the additional confirmation slot in each phase, the running time of this algorithm is again $O(\log n + \Delta^2)$ *w.h.p.*

Algorithm 6: A Las Vegas 2-hop-colouring algorithm in $B_{cd}L_{cd}$ (without knowledge).

```
Float p ← 1/2;
Integer colour ← nil;
Integer counter ← 0;
repeat
    switch slot do
        slot 1 // contending slot
            if colour = nil then
                beep with probability p;
        slot 2 // peripheral collision detection (and consequences)
            if several neighbours beeped in slot 1 then
                beep
            if I beeped alone in slot 1 and heard no beep in slot 2 then
                colour ← counter
        slot 3 // adaptive probability
            if someone beeped in slot 1 then
                beep
            if colour = nil then
                if no beep heard in slot 1 nor 3 then
                    increase p
                else
                    decrease p
        slot 4 // termination slot
            if colour = nil then
                beep
    counter ← counter + 1
until no beep heard in slot 4;
```

4.4 Jeavons et al.'s Las Vegas Algorithm for the MIS in $B_{cd}L$

We recall here Jeavons et al.'s Las Vegas Algorithm for the MIS [8]. This algorithm uses an *adaptive probability* to maximize the frequency of exclusive beeps (with a doubling/halving pattern for p, starting at $1/2$). If a node v produces an exclusive beep, it enters the MIS (by the end of the first slot), then it uses a confirmation slot to inform its neighbours, all of which terminate together with v. Since the whole neighbourhood shuts down at once, the algorithm progresses faster than, for instance, the basic colouring algorithm discussed above. This algorithm was already proven by Jeavons et al. to terminate within $O(\log n)$ slots with a huge constant factor (larger than e^{25}).

4.5 Computing a 2-hop MIS

In this problem, we must select a set of nodes (the MIS) such that no pair of selected nodes are within distance 2 and no node can be added further to the set. This algorithm is a combination of those of other 2-hop algorithms seen above, and Jeavons et al's MIS algorithm. That is, the same structure of algorithm is used as for 2-hop colouring or degree computation, except that whenever a node produces a 2-hop exclusive beep, it enters the MIS and informs its neighbours (using the confirmation slot) that they will not be in the MIS. This algorithm

takes the same number of *phases* than what the (1-hop) MIS algorithm would produce in the square of the graph, that is, $O(\log n)$ *w.h.p.*. The number of slots is higher due to using additional slots for managing 2-hop exclusive beeps, but it remains within a (small) constant factor. Interestingly, our analysis of this algorithm improves much over that of [8], taking the huge constant down to 76 (i.e., making the algorithm practical).

5 Collision detection and emulation techniques

In Section 4, we have considered collision detection as a built-in primitive. Depending on the algorithms, we assumed that collision detection was possible while beeping (B_{cd}) or while listening (L_{cd}). This assumption is convenient because it allows one to design *Las Vegas* algorithms for all the considered problems. Unfortunately, we know since [1] that no Las Vegas algorithms can be designed for most problems without collision detection, that is, in the BL model. One has to turn to Monte Carlo instead, which means that the result is correct only with some probability (possibly *w.h.p.*). In this section, we investigate the cost of building a probabilistic collision detection primitive in the BL model, inspired by a technique from [1]. Then we adapt it into two generic emulation procedures, one for detecting collision while beeping, the other while listening. These procedures can then be used to translate any Las Vegas algorithml in $B_{cd}L$, BL_{cd}, or $B_{cd}L_{cd}$, into a Monte Carlo algorithm in BL. The cost is a logarithmic slowdown of the execution, which we prove is essentially optimal (for sufficiently large n).

5.1 Collision detection

The impossibility for a node in BL to distinguish between begin alone or having neighbours has strong implications. For instance, in the colouring problem, it means that two neighbours could possibly end up with the same colour. In the MIS problem, two neighbours could enter the MIS. In fact, there is no guarantee on the correctness of basic patterns like exclusive beeps or 2-hop exclusive beeps, which are at the basis of most (if not all) Las Vegas algorithms.

We present a (Monte Carlo) algorithm for detecting collisions in BL. This procedure generalises the technique used in Algorithm 3 of [1], which consists of replacing each slot that requires collision detection in the original model, with several BL slots in which symmetries are probabilistically broken. Of course, the more slots, the more reliable the detection.

The algorithm

Each slot that requires collision detection (B_{cd} or L_{cd}) is replaced with a number of *sub-phases*, each consisting of two BL slots. For instance, if a node wishes to beep with collision detection in the original algorithm, it will choose one of the two slots (*u.a.r.*) in each of the sub-phases and will beep in that slot (listen in the other). If it hears a beep while listening in the other slot, then an internal collision is detected. Similarly, if a node wishes to listen with collision detection in the original algorithm, it will listen in both slots of each sub-phase. A peripheral collision is detected if a beep is heard in both slots of a same sub-phase. The procedure is detailed by Algorithm 7, where k is the number of sub-phases used.

False positives never happen, but real collisions might still go unnoticed, with probability inversely related to k. We are interested in determining how large k should be to guarantee that a given node detects a collision in its neighbourhood with a given probability. The stronger question asks how many sub-phases are required to guarantee that *none* of the nodes fails to detect a collision.

Algorithm 7: Collision detection algorithm in BL (with parameter k)

```
Boolean collision ← false;
Integer i ← 0;
while i < k do
    if v wishes to beep then
        Flip a coin;
        if heads then
            beep in slot 1;
            listen in slot 2;
        else
            listen in slot 1;
            beep in slot 2;
        if another beep was heard then
            collision ← true
    else
        listen in both slots;
        if beeps are heard in both slots then
            collision ← true;
    i ← i + 1;
return collision;
```

▶ **Lemma 1.** *Let v be a node. If a collision occurs in the neighbourhood of v, then v detects it in $O\left(\log(\frac{1}{\epsilon})\right)$ sub-phases (slots) with probability at least $1-\epsilon$, and in $O(\log n)$ sub-phases (slots) with probability $1-o\left(\frac{1}{n^2}\right)$.*

Proof. Assume a collision occurs between some nodes u_1 and u_2 in the neighbourhood of v (one of them being possibly v itself). It is detected if u_1 and u_2 choose a different slot in at least one of the k sub-phases. The probability that this does not happen is $\left(\frac{1}{2}\right)^k$. This probability is less than ϵ (resp. $o\left(\frac{1}{n^2}\right)$) for any $k \geq \log(\frac{1}{\epsilon})$ (resp. $2\log(n)$). Observe that if collisions occur between more than two nodes in the neighbourhood of v, this cannot decrease the odds of a successful detection (to the contrary, the odds can only increase). ◀

▶ **Corollary 2.** *Let G be a graph. If collisions occur in the neighbourhood of an arbitrary number of nodes, then all of them detect collision after at most $O\left(\log(\frac{n}{\epsilon})\right)$ sub-phases (slots) with probability at least $1-\epsilon$, and after at most $O(\log n)$ sub-phases (slots) w.h.p.*

Proof. Assume collisions occur in G and let T denote the number of sub-phases before all concerned nodes detect collision. Clearly $T = \max\{T_v \mid v \in V\}$, where T_v is the time it takes to any node v to decide collision. By the same argument as in the proof of Lemma 1, together with union bound, it holds that

$$\mathbb{P}r\left(T > \log\left(\frac{n}{\epsilon}\right)\right) \leq n \times \mathbb{P}r\left(T_v > \log\left(\frac{n}{\epsilon}\right)\right) \quad (1)$$

$$= n \times \frac{1}{2^{\log(\frac{n}{\epsilon})}} = \epsilon \quad (2)$$

which proves the first claim. The same argument, combined with the second claim of Lemma 1 proves the second claim. ◀

Algorithm 8: A Procedure to emulate a B_{cd} in the BL model.

```
Procedure EmulateL_cdinBL(in : Integer k, Array s; out : Boolean collision)
Boolean collision ← false;
Integeri ← 0;
repeat
    if s[i] then beep in slot 1; listen in slot 2;
    else listen in slot 1; beep in slot 2;
    if another beep was heard then collision ← true;
    i ← i + 1
until i = k;
End Procedure
```

Algorithm 9: A Procedure to emulate a L_{cd} in the BL model.

```
Procedure EmulateL_cdinBL(in : Integer k; out : Boolean beep, Boolean collision)
Boolean beep ← false;
Boolean collision ← false;
Integer i ← 0;
repeat
    switch slot do
        slots 1 and 2
            listen
        end of phase:
            if a beep was heard in any slot then
                beep ← true
            if a beep was heard in both slots then
                collision ← true
    i ← i + 1
until i = k;
End Procedure
```

5.2 Emulation procedures

Based on this tie-breaking mechanism, we define two probabilistic emulation procedures whose purpose is to replace beep or listen instructions with collision detection in BL. Both are Monte Carlo in the sense that detection is only guaranteed with some probability. The first procedure, `EmulateB`$_{cd}$`inBL()`, is given by Algorithm 8 and the second, `EmulateL`$_{cd}$`inBL()`, by Algorithm 9. Both procedures are parametrized by an integer $k > 1$, which accounts for the number of sub-phases that are used in each invocation of the procedure (k controls the error bound). They return `true` if a collision has been detected, `false` otherwise.

Before the execution each vertex generates a sequence s of k random bits (*u.a.r.*) which will be the ones used in each sub-phase. The reason why this is made once at the beginning rather than in each invocation is a technicality that relates to preventing an additional union bound in the analysis (more k would be needed to guarantee that *each* invocation is successful if the numbers are drawn every time).

Hence, the value of k depends on the bound we require on the probability of error, a straightforward adaptation of the above analysis gives us the values of Lemma 3.

▶ **Lemma 3.** *For any $\varepsilon > 0$, and any $n > 0$:*

1. *if $k = \lceil \log\left(\frac{1}{\varepsilon}\right) \rceil$, the procedures are correct for a given node with probability $1 - \varepsilon$;*
2. *if $k = \lceil \log\left(\frac{n}{\varepsilon}\right) \rceil$, the procedures are correct for any node with probability $1 - \varepsilon$;*
3. *if $k = \lceil 2\log(n) \rceil$, the procedures are correct for any node w.h.p.*

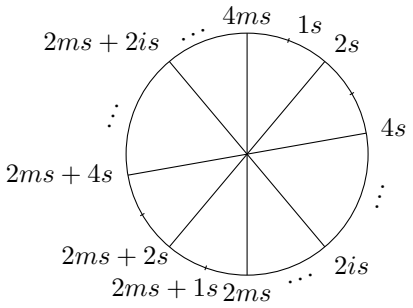


Figure 1 The wheel gadget used in the proof of optimality for emulation.

Observe that in general, the size of the network n is not known to the nodes, which is an obstacle to achieving the second and third types of guarantees. However, it is reasonable in practice to assume that the nodes know an *upper bound* on n, e.g., when a network of wireless sensors is deployed. The upper bound may even be loose without much consequence: so long as it is polynomial in n, the slowdown factor remains of the same order.

Using the procedures

In the listings of our algorithms (see Section 4), listen instructions are implicit. By default, a node listens if it does not beep. Emulation procedures should be used explicitly for both *beep* and *listen* primitives, in order for the nodes to remain synchronized (since each of them takes logarithmically many rounds to be carried out). Therefore, whenever a node calls `EmulateB`$_{cd}$`inBL` or `EmulateL`$_{cd}$`inBL`, the other nodes should call one of these or wait the corresponding amount of time. Likewise, the procedures should not be interrupted even after a collision with a given neighbor is detected, to preserve synchrony with other neighbours or farther nodes.

5.3 Optimality of the emulation

In this section we prove that the emulation procedures presented in Section 5.2 are essentially optimal (i.e. asymptotically and up to a constant factor), namely, we prove a $\Omega(\log n)$ lower bound on the number of slots required to detect collision in some graphs called *wheels*. A (m, s)-wheel, illustrated in Figure 1, is a graph $W = (V, E)$ such that $V = u_1, \dots u_{4ms}$, the edges E are all the (u_{i-1}, u_i) (arithmetic modulo $4ms$) plus m spokes, that is edges $(u_{is}, u_{(i+2m)s})$ $(1 \leq i \leq 2m)$, where the wheel can be odd (all spokes with i odd) or even (all spokes with i even). The even and odd (m, s)-wheels are isomorphic. We consider only situations in which all vertices u_{is} are in the same state, a state in which they wish to beep and all other vertices are in the same internal state, a state in which they do not wish to beep. Thus vertices at the ends of spokes and no others must conclude that there is a collision. The slot complexity of any algorithm which detects collision in such a graph with high probability is to be $\Omega(\log n)$. Due to space limitations, the full proofs are relegated to the long version. We provide, however, a minimal sentence of insight for each.

Considering a computation of a collision detecting algorithm on a wheel, we define, for any $t > 0$, b_t^i as the signal (beep or not) from u_i to all its neighbours at time t, and, for any $t \geq 0$, B_t^i the sequence $b_1^i \cdots b_t^i$. Then, we define the event E_t for a spoke $u_{is}, u_{(i+2m)s}$ as follows:

$$E_t = \left\{ (B_t^{is} = B_t^{(i+2m)s}) \wedge (B_t^{is+1} = B_t^{(i+2m)s+1}) \wedge (B_t^{is-1} = B_t^{(i+2m)s-1}) \right\}.$$

▶ **Lemma 4.** *For any t $(0 \leq t < s)$, it holds that $\mathbb{P}r(E_t) \geq 2^{-3t}$.*

The proof proceeds by induction on t, with base case $t = 0$. (Full proof in the long version.)

If E_t holds for the spoke $(u_{is}, u_{(i+2m)s})$, we say that the spoke fails to break symmetry within time t. This happens with probability at least 2^{-3t} and, if it happens, the existence of the spoke has had no influence on the computation up to time t. In particular, whenever u_{is} beeped, $u_{(i+2m)s}$ also beeped and so neither has ever heard the other beep.

▶ **Theorem 5.** *For any Monte Carlo algorithm $\mathcal{A}$ which detects collision in W, if $\mathcal{A}$ halts in less than $\log_2 n/4$ rounds with probability greater than $3/4$ then for some situations in some wheels, $\mathcal{A}$ gives incorrect results for some vertices with probability greater than $1/4$.*

The proof proceeds using the wheel gadget of Figure 1 and Lemma 4 to characterize the rate at which the symmetry induced by the spokes can be broken. (Full proof in the long version.)

▶ **Corollary 6.** *The complexity of a Monte Carlo algorithm which detects collision with high probability in the BL model is $\Omega(\log n)$.*

References

1. Y. Afek, N. Alon, Z. Bar-Joseph, A. Cornejo, B. Haeupler, and F. Kuhn. Beeping a maximal independent set. *Distributed Computing*, 26(4):195–208, 2013.
2. K. Azuma. Weighted sums of certain dependent random variables. *Tohoku Mathematical Journal, Second Series*, 19(3):357–367, 1967.
3. A. Casteigts, Y. Métivier, J. Michael Robson, and A. Zemmari. Counting in one-hop beeping networks. *CoRR*, abs/1605.09516, 2016.
4. J. Collier, N. Monk, P. Maini, and J. Lewis. Pattern formation by lateral inhibition with feedback: a mathematical model of delta-notch intercellular signalling. *Journal of Theoretical Biology*, 183(4):429–446, 1996.
5. A. Cornejo and F. Kuhn. Deploying wireless networks with beeps. In *DISC*, pages 148–162, 2010.
6. S. Gilbert and C. Newport. The computational power of beeps. In *Proc. of 29th International Symposium on Distributed Computing (DISC)*, 2015.
7. B. Huang and Th. Moscibroda. Conflict resolution and membership problem in beeping channels. In *DISC*, pages 314–328, 2013.
8. P. Jeavons, A. Scott, and L. Xu. Feedback from nature: simple randomised distributed algorithms for maximal independent set selection and greedy colouring. *Distributed Computing*, 2016. doi:10.1007/s00446-016-0269-8.
9. S. Navlakha and Z. Bar-Joseph. Distributed information processing in biological and computational systems. *Commun. ACM*, 58(1):94–102, 2015.
10. J. Schneider and R. Wattenhofer. What is the use of collision detection (in wireless networks)? In *DISC*, pages 133–147, 2010.
11. A. Scott, P. Jeavons, and L. Xu. Feedback from nature: an optimal distributed algorithm for maximal independent set selection. In *PODC*, pages 147–156, 2013.

Collision-Free Pattern Formation*

Rachid Guerraoui[1] and Alexandre Maurer†[2]

1 **EPFL, Lausanne, Switzerland**
`rachid.guerraoui@epfl.ch`
2 **EPFL, Lausanne, Switzerland**
`alexandre.maurer@epfl.ch`

Abstract

Shoals of small fishes can change their collective shape and form a specific pattern. They do so efficiently (in parallel) and without collision.

In this paper, we study the analog problem of distributed pattern formation. A set of processes needs to move from a set of initial positions to a set of final positions. The processes are oblivious (no internal memory) and must preserve, at any time, a minimal distance between them.

A naive solution would be to move the processes one by one, but this would take too long. The difficulty here is to move the processes simultaneously in clearly delimited phases, no matter how unfavorable the initial configuration may be. We solve this by treating the problem "dimension by dimension": the processes first form 1D trails, then gather into a 2D shape (this technique can be generalized to higher dimensions).

We present an optimal algorithm which time complexity depends linearly on the radius of the smallest circle containing both initial and final positions. The algorithm is self-stabilizing, as the processes are oblivious and the initial positions are arbitrary.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Pattern formation, Collision, Landmarks

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.16

1 Introduction

Nature has always been a fertile source of inspiration for computing: evolutionary algorithms [2], neural networks [16] or cellular automata [22] are all based on natural phenomena.

One remarkable phenomenon is the ability of simple individuals (ants, bees, fishes ...) to form complex patterns without centralized control. For instance, shoals of fishes can form a massive compact structure [20], which can then deform itself to cross small holes and avoid predators. These individuals form complex patterns very efficiently by moving to the new positions in parallel. They do so without accident: they do not even touch each other. Reproducing this phenomenon has a particular interest in the era of autonomous mobile devices [18]. This could have many applications, such as the exploration of dangerous or impracticable zones [3] as well as medical nanorobots [19].

In this paper, we consider the problem of arbitrary pattern formation with landmarks [14]: a set of n processes, with a set of n arbitrary initial positions, must move to a set of n arbitrary final positions (*landmarks*) forming some desired pattern. The difficulty of this

* This work has been supported in part by the Swiss National Science Foundation, grant 200021_169588 / TARBDA.
† Contact author.


problem lies on the requirement to consider "voluminous" processes (namely processes do have a volume). More specifically, a minimal *distance D* between any two processes needs to be preserved at any time. This requirement is essential for any application where the volume of processes is not negligible. Besides, many types of devices are fragile (quadcopter drones, robots exploring Mars ...) and collisions could render them unusable.

Most existing works on pattern formation [21, 8, 10, 23] ignore collisions: each process is represented by a point, and can move independently of other processes. Several processes can thus occupy the exact same position. Pattern formation has indeed been considered without collisions [13, 14, 15], but two processes could be as close as possible to each other, as long as they do not occupy the exact same point in space. These approaches do not apply to the setting where the volume of processes imposes a minimal distance between any two processes. Some papers considered problems where processes have a certain volume, such as localization [6], gathering [5, 17, 4, 1], circle formation [7] or coating [9]. However, the more general problem of arbitrary pattern formation has never been studied for voluminous processes.

We study this problem in a general setting where processes are *oblivious* (they have no form of internal memory) and cannot communicate with each other by message passing or signals. The only form of "communication" lies in the ability of each process to see the position of other processes (we do not put restrictions on their visibility). A naive solution would be to move the processes far away from the final positions, then to have the processes move one by one to the desired positions. Clearly, such a solution would be extremely ineffective in terms of execution time for a large number of processes (ideally, the processes should all move together).

The difficulty here is to separate the moves in clearly delimited *phases*, that can be identified without ambiguity only by looking at the positions of the processes. Besides, this should be done while all processes move simultaneously *and* with the constraint of respecting a minimal distance at any time, no matter how the processes are arranged in the initial configuration (i.e., possibly in a very unfavorable way).

In this paper, we present a solution which execution time depends only linearly on the distance between initial and final positions, which we show is optimal in terms of time complexity. Our algorithm involves two major steps:

- First, we provide a sub-algorithm to form *trails* without collisions (Collision-Free Trail Formation, CFTF). A trail is a slice of pattern contained in a very tight band. The idea underlying this algorithm is for the processes to scatter, align orthogonally to the band and finally gather.
- Then, we provide an algorithm that uses the CFTF algorithm as a subroutine (Collision-Free Pattern Formation, CFPF). This algorithm slices the desired pattern in several trails, scatters the trails in the orthogonal direction, uses the CFTF algorithm to form the desired trails, and finally gathers the trails to obtain the desired pattern.

The main idea here is to treat the problem "dimension by dimension". For presentation simplicity, we consider the pattern formation problem in a 2D space, but the same principle could be repeated to reach higher dimensions : in the same way that we stack "trails" to form a 2D pattern, we could stack "slices" very similar to 2D patterns to form any 3D pattern – and so forth.

Besides, our resulting CFPF algorithm is *self-stabilizing* [11], as the processes are oblivious and the initial positions are totally arbitrary. Thus, any transient disruption simply brings the system to a new initial configuration.

We also prove that this algorithm has an optimal time complexity. We express the time complexity as a function of R: the radius of the smallest circle containing both initial and

final positions of the processes. This is in contrast to classical time complexity that is often expressed as an asymptotic function of n (the total number of processes), which is not sufficient here: the execution time is conditioned by the initial and final distances between processes. The time complexity of the naive solution where the processes move one by one is $O(R^3)$ (see Section 2.3). We show that the time complexity of the pattern formation problem without collision is only $\Theta(R)$, and that our algorithm matches this bound (i.e., a tight bound on the time complexity is $\Omega(R)$).

The rest of the paper is organized as follows. In Section 2, we describe the problem, the model and our time complexity criteria (we show that this complexity is $\Omega(R)$), then we give an overview of our solution. In Section 3, we present our Collision-Free Trail Formation (CFTF) algorithm and show its $O(R)$ complexity. In Section 3, we present our Collision-Free Pattern Formation (CFPF) algorithm and show its $O(R)$ complexity. We conclude in Section 5.

2 Preliminaries

2.1 Model

We consider a 2D space with a (x, y) coordinates system, and a set P of n processes. Each process is described by its (x, y) position. We denote by $d(A, B)$ (resp. $d(p, q)$) the euclidean distance between two points A and B (resp. two processes p and q) in the space[1]. The processes must observe a minimal distance D between each other: for any two processes p and q, we must have $d(p, q) \geq D$.

We adopt the classical FSYNC (*fully synchronous*) model for swarm computing [12]. The time is divided into successive steps $t = 0, 1, 2, 3 \ldots$ Let $\epsilon > 0$ be an arbitrarily small constant. ϵ is the largest distance that a process can cross between two time steps. At each time t, each process p uses its position and the position of other processes to compute a point M such that $d(p, M) \leq \epsilon$, and moves to this point ("Look–Compute–Move"). Then, the position of p at time $t + 1$ will be M.

The processes are anonymous (one can distinguish two processes only by their position) and oblivious (no form of memory is available).

Note 1: In this problem, the processes are required to know n positions to reach in the plane (also called *landmarks* [14]). Thus, a common coordinates system is a legitimate hypothesis, as the processes can also know three specific points of the plane that provide origin, metric and orientation. In practice, as each process is assumed to see other processes, these three points could also be obtained from elements of the setting.

Note 2: We adopt a synchronous model for the clarity of presentation. It should not be seen as a succession of discrete steps where the processes "teleport" between each step, but rather as the approximation of a continuous move: the maximal distance ϵ that a process can cross between two steps can be as small as we want. In this paper, we focus on avoiding collisions during the pattern formation – which remains nontrivial even if we abstract symmetry or synchrony problems[2]

[1] If A (resp. B) has the coordinates (x, y) (resp. (x', y')), then $d(A, B) = \sqrt{(x - x')^2 + (y - y')^2}$.

[2] In [15], it was shown that a pattern formation problem solvable in the synchronous model was also solvable in the asynchronous model. However, it does not say anything about preserving a minimal distance between processes during the whole execution.

2.2 Problem

Let S be an arbitrary set of n points such that, for any two points A and B of S, $d(A, B) \geq D$. At $t = 0$, the n processes have an arbitrary position such that, for any two processes p and q of P, $d(p, q) \geq D$. The problem consists of finding an algorithm satisfying the two following conditions:

- **Liveness.** The n processes always eventually occupy the n positions of S permanently (and then do not move from these positions).
- **Safety.** At any point in time, the condition on the minimal distance between processes must always be respected: at any time and for any two processes p and q, $d(p, q) \geq D$.

For simplicity reasons, we only consider collisions at each discrete step. However, this is not a problem in practice, as ϵ can be as small as we want. To ensure that there is no collision between two steps, a simple solution is to impose a minimal distance $D + \epsilon$ instead of D. Also, note that even without this hypothesis, our algorithm prevents any collision between two steps (assuming that the processes move in straight line).

2.3 Time complexity

The performance of a pattern formation algorithm can be measured by its *time complexity*, that is: the asymptotic behavior of its execution time, i.e. the number of time steps required to form the pattern.

The time complexity of an algorithm is often expressed as a function of n (the total number of processes). However, this criteria is not sufficient in the case of mobile processes with a bounded speed: the distance between the initial and final positions imposes a minimal execution time, independently of n. In other words, it is impossible to bound the execution time with a function of n: if such a bound existed, it would always be possible to overstep it by increasing the distance between initial and final positions sufficiently.

Therefore, we consider another parameter here: R, the radius of the smallest circle that contains all initial and final positions of the processes. More precisely, R is the smallest number for which there exists a point O such that:

1. At $t = 0$, for any process p, $d(O, p) \leq R$.
2. For any point M of S, $d(O, M) \leq R$.

Note that, as there is a minimal distance between processes, R implicitly imposes a bound on n: the maximal number of processes that we can put into a circle of radius R with a minimal distance D between processes. In other words, the number of processes n can be proportional to R^2.

We want to express the time complexity T of the problem (that is, the number of steps to form the pattern) as a function of R. For this purpose, we use the classical asymptotic Landau notation: Ω, O and Θ.

- T is "$\Omega(f(R))$" if there exists R_0 and $k > 0$ such that $\forall R \geq R_0$, $T \geq kf(R)$. This is a *lower* bound: the time complexity is *at least* $kf(R)$ for a sufficiently large R.
- T is "$O(f(R))$" if there exists R_0 and $k > 0$ such that $\forall R \geq R_0$, $T \leq kf(R)$. This is an *upper* bound: the time complexity is *at most* $kf(R)$ for a sufficiently large R.
- T is "$\Theta(f(R))$" if T is both $\Omega(f(R))$ and $O(f(R))$ (*tight* bound). In other words, there exists R_0, $k_1 > 0$ and $k_2 > 0$ such that $\forall R \geq R_0$, $k_1 f(R) \leq T \leq k_2 f(R)$.

The time complexity of an algorithm where the processes move "one by one" (like in [4]) is at least proportional to nR. Thus, as n can be proportional to R^2, such an algorithm only

provides an upper bound $O(R^3)$ to the time complexity. In this paper, we show that the time complexity of the CFPF problem is only $\Theta(R)$:

- The lower bound is trivial. Consider the case where the n processes are on a circle of radius R, and that the center of this circle is one of the final positions. Then, the execution time is at least R/ϵ (the minimal time for a process to reach the center of the circle). Thus, the time complexity of the problem is $\Omega(R)$.
- For the upper bound, we provide an algorithm solving the CFPF problem with an execution time $O(R)$. We present this algorithm in the next section.

2.4 Overview of our algorithm

We give here an intuition of how we construct an algorithm to solve the CFPF problem in $O(R)$ time steps.

We first define the notion of a *trail*. Intuitively, a trail is a set of positions which is arbitrarily large in the x direction, but very tight in the y direction.

▶ **Definition 1** (Trail). For any point or process A, let $x(A)$ (resp. $y(A)$) be the x (resp. y) coordinate of A. A *trail* is a set of points T such that:

1. For any two points A and B of T, $d(A, B) \geq D$.
2. The "width" of T in the y direction is at most $D/2$: for any two points A and B of T, $|y(A) - y(B)| \leq D/2$.

Then, the set S of final positions can be seen as several trails stacked in the y direction, as shown below.

Let y_1 (resp. y_2) be the smallest (resp. largest) y coordinate occupied by a point of S. Let N be the smallest integer such that $y_1 + ND/2 > y_2$. $\forall i \in \{1, \ldots, N\}$, let T_i be the set of points M of S such that $y_1 + (i-1)D/2 \leq y(M) < y_1 + iD/2$. According to Definition 1, T_i is a trail. Then, the set S of final positions can be decomposed in the set of trails $\{T_1, \ldots, T_N\}$.

In Section 3, we give an algorithm to form any trail (Collision-Free Trail Formation algorithm), and bound its execution time. Then, in Section 4, we use this algorithm as a subroutine to define a CFPF algorithm with a $O(R)$ time complexity. This algorithm forms the trails $\{T_1, \ldots, T_N\}$, then stacks them in the y direction.

3 Collision-Free Trail Formation (CFTF) Algorithm

In this section, we give a CFTF algorithm and bound its execution time. We first define the CFTF problem, introduce some definitions and primitives, and give an informal description of the algorithm. Then, we describe the algorithm and show its execution time.

3.1 CFTF Problem

Consider a set Q of m processes[3]. Let T be a *trail* (see Definition 1) such that $|T| = m$. At $t = 0$, the m processes have an arbitrary position such that, for any two processes p and q of Q, $d(p, q) \geq D$. We want to find an algorithm such that the m processes always eventually occupy the m positions of T (*liveness*). During the whole process, the condition on the minimal distance between processes must always be respected: at any time and for any two processes p and q, $d(p, q) \geq D$ (*safety*).

[3] We use m instead of n to make a clear distinction between the CFTF problem and the CFPF problem.

3.2 Definitions

In order to describe our CFTF algorithm, we define the minimal x coordinate of the processes (resp. positions), define a total order relationship on the processes (resp. positions), and finally the elementary distances between processes (resp. positions) according to this order.

Minimal x position. Let $x_{\min}$ be the largest number such that, for each process p, $x(p) \geq x_{\min}$. In other words, $x_{\min}$ is the smallest x coordinate occupied by a process. Similarly, let x_0 be the smallest x coordinate occupied by a point of T.

Total order relationship. We define a total order relationship "$>$" on the position of processes. We say that $p > q$ if one of the two following conditions is satisfied:
(1) $x(p) > x(q)$,
(2) $x(p) = x(q)$ and $y(p) > y(q)$.

Note that the order relationship is total, as the condition on the minimal distance between processes forbids two processes to be at the same position. We also define a similar order relationship on the points of T.

For any time t, let $(p_1, p_2, \ldots, p_m)$ be the m processes such that $p_1 < p_2 < \cdots < p_m$. Let $(M_1, M_2, \ldots, M_m)$ be the m points of T such that $M_1 < M_2 < \cdots < M_m$.

Elementary x distances. $\forall i \in \{1, \ldots, m-1\}$, let $d_i = x(p_{i+1}) - x(p_i)$ (distances between processes) and $D_i = x(M_{i+1}) - x(M_i)$ (distances between positions).

3.3 Primitives

We define here some basic primitives used in the algorithm to describe the moves of the processes. Let p be a process with coordinates (x, y) at time t, and let M be a point of same coordinates.

"Move towards" primitive. Let M' be a point. Let K be a point of the segment $[MM']$ such that $d(M, K) = \epsilon$. If such a point does not exist (that is, if $d(M, M') < \epsilon$), we consider that $K = M'$. Then, we say that p *moves towards* M' if the position of p at time $t + 1$ is K.

"Move with vector" primitive. Let $[a, b]$ be a $2D$ vector such that $\sqrt{a^2 + b^2} \leq \epsilon$. Let K be a point of coordinates $(x + a, y + b)$. Then, we say that p *moves with vector* $[a, b]$ if the position of p at time $t + 1$ is K.

3.4 CFTF Algorithm

Our CFTF algorithm is described in Figure 1. We provide an informal description of the algorithm below.

The algorithm goes through 4 successive phases: the processes translate to the good x offset (Phase 1), increase their x distance between each other (Phase 2), align in the y direction (Phase 3) and finally adjust their distance in the x direction until they form the desired trail.

1. In Phase 1, the processes translate in the x direction until the smallest x coordinate occupied by a process ($x_{\min}$) corresponds to the smallest x coordinate of the set of final positions (x_0).

CFTF Algorithm

At each step, every process p executes the following algorithm, which is divided in 4 phases. The conditions of the 4 phases are defined to be mutually exclusive.

Predicates

- $P_1 : x_{\min} = x_0$
- $P_2 : \forall i \in \{1, \ldots, m-1\},\ d_i \geq \max(D, D_i)$.
- $P_3 : \forall i \in \{1, \ldots, m\},\ y(p_i) = y(M_i)$.
- P_4 : The m processes occupy the m positions of T.

Phase 1: Translation

Condition: $\neg P_1$

Action: If $|x_{\min} - x_0| < \epsilon$, move with vector $[x_0 - x_{\min}, 0]$. Otherwise:
- If $x_{\min} > x_0$, move with vector $[-\epsilon, 0]$.
- If $x_{\min} < x_0$, move with vector $[\epsilon, 0]$.

Phase 2: Scattering

Condition: $P_1 \wedge \neg P_2 \wedge \neg P_3$

Action: Let i be the smallest integer such that $d_i < \max(D, D_i)$.
Then, if p belongs to $\{p_{i+1}, p_{i+2}, \ldots, p_m\}$, move with vector $[\epsilon, 0]$.

Phase 3: Alignment

Condition: $P_1 \wedge P_2 \wedge \neg P_3$

Action: Let i be such that $p = p_i$. Let M be the point of coordinates $(x(p_i), y(M_i))$.
Then, move towards M.

Phase 4: Gathering

Condition: $P_1 \wedge P_3 \wedge \neg P_4$

Action: Let i be the smallest integer such that $d_i > D_i$. Let $e = d_i - D$.
Then, if p belongs to $\{p_{i+1}, p_{i+2}, \ldots, p_m\}$:
- If $e \leq \epsilon$, move with vector $[-e, 0]$.
- If $e > \epsilon$, move with vector $[-\epsilon, 0]$.

Figure 1 CFTF Algorithm.

2. In Phase 2, the processes scatter in the x direction until we have $d_i \geq \max(D, D_i)$ for each i. $d_i \geq D$ ensures that the processes can move in the y direction without collision in the next phase. $d_i \geq D_i$ ensures that the x distance between processes is at least the desired distance D_i. Thus, the processes only have to reduce this distance in Phase 4 to obtain the desired trail.
3. In Phase 3, the processes move in the y direction until they all have the desired y position. As we have $d_i \geq D$ for each i, there is no collision between processes during this phase.
4. In Phase 4, the processes gather in the x direction until we have $d_i = D_i$ for each i. According to Phase 2, they only have to reduce their distance between each other. Then, we have the desired x and y positions, and thus the desired trail.

3.5 Time complexity

Similarly to R (see Section 2.3), let r be the radius of the smallest circle containing all initial and final positions of the m processes. In Theorem 3, we show that the execution time of the CFTF algorithm is $O(r)$. For this purpose, we first bound m with r in Lemma 2.

▶ **Lemma 2.** $m \leq 1 + 2r/D$

Proof. First, let us show that $\forall i \in \{1, \ldots, m-1\}$, $D_i \geq D/2$. Indeed, assume the opposite: there exists i such that $D_i < D/2$. Then, there exists two final positions A and B such that $|x(A) - x(B)| \leq D/2$. As $|y(A) - y(B)| \leq D/2$, we have $d(A, B) \leq D/\sqrt{2} < D$: contradiction.

Therefore, $\Sigma_{i=1}^{i=m-1} D_i \geq (m-1)D/2$. As $\Sigma_{i=1}^{i=m-1} D_i \leq r$ (by definition), we have $(m-1)D/2 \leq r$. Thus, $m \leq 1 + 2r/D$. ◀

▶ **Theorem 3.** *The m processes occupy the m desired positions in $O(r)$ time steps, and the condition on the minimal distance is always respected.*

Proof. The proof is in 4 steps.

1. Suppose $\neg P_1$. The distance of translation in Phase 1 is at most r. Thus, P_1 is true after a time $T_1 \leq \lceil r/\epsilon \rceil = O(r)$. As the translation is the same for all processes, the safety condition is respected.
2. Suppose $P_1 \wedge \neg P_2 \wedge \neg P_3$. Let i be the integer described in Phase 2. Then, we have $d_i \geq \max(D, D_i)$ in at most $\lceil \max(D, D_i)/\epsilon \rceil$ time steps. Thus, $P_1 \wedge P_2 \wedge \neg P_3$ is true after a time $T_2 \leq \Sigma_{i=1}^{i=m-1} \lceil \max(D, D_i)/\epsilon \rceil \leq \Sigma_{i=1}^{i=m-1} \lceil D/\epsilon \rceil + \Sigma_{i=1}^{i=m-1} \lceil D_i/\epsilon \rceil \leq m(1 + D/\epsilon) + (m + \Sigma_{i=1}^{i=m-1} D_i/\epsilon) \leq (2 + D/\epsilon)m + r/\epsilon$. According to Lemma 2, $m \leq 1 + 2r/D$, and thus, $T_2 \leq 2 + D/\epsilon + (4/D + 3/\epsilon)r = O(r)$. As the moves of Phase 2 only increase the distance between processes, the safety condition is respected.
3. Suppose $P_1 \wedge P_2 \wedge \neg P_3$. The moves of Phase 1 and 2 do not modify the y position of the processes. Thus, $\forall i \in \{1, \ldots, m\}$, $|y(p_i) - y(M_i)| \leq r$. Therefore, in Phase 3, $P_1 \wedge P_3$ is true after a time $T_3 \leq \lceil r/\epsilon \rceil = O(r)$. According to P_2, for any two processes p and q, $|x(p) - x(q)| \geq D$. Thus, as the moves of Phase 3 are only in the y direction, the safety condition is respected.
4. Suppose $P_1 \wedge P_3 \wedge \neg P_4$. $\forall i \in \{1, \ldots, m-1\}$, Phase 2 increases d_i of at most $\max(D, D_i) + \epsilon$. Let Δ_i be the value of d_i at the beginning of Phase 4. Then, $\Sigma_{i=1}^{i=m-1} \Delta_i \leq r + \Sigma_{i=1}^{i=m-1} (\max(D, D_i) + \epsilon) \leq r + m(D + \epsilon) + \Sigma_{i=1}^{i=m-1} D_i \leq r + (1 + 2r/D)(D + \epsilon) + r \leq 2(2 + \epsilon/D)r + D + \epsilon$. Let i be the integer described in Phase 4. Then, we have $d_i = D_i$ in at most $\lceil \Delta_i/\epsilon \rceil \leq 1 + \Delta_i/\epsilon$ time steps. Thus, P_4 is true after a time $T_4 \leq \Sigma_{i=1}^{i=m-1} (1 + \Delta_i/\epsilon) \leq m + (\Sigma_{i=1}^{i=m-1} \Delta_i)/\epsilon \leq 1 + 2r/D + (2(2 + \epsilon/D)r + D + \epsilon)/\epsilon = 4(1/D + 1/\epsilon)r + 2 + D/\epsilon = O(r)$. As the moves of Phase 4 do not allow to have $d_i < D_i$, the safety condition is respected.

Once P_4 is true, the processes do not move, and thus the liveness condition is satisfied. The execution time is at most $T_1 + T_2 + T_3 + T_4 = O(r)$. ◀

4 Collision-Free Pattern Formation (CFPF) Algorithm

In this section, we use our previous CFTF algorithm to define a CFPF algorithm and bound its execution time. Similarly, we start with some definitions and an informal description of the algorithm.

4.1 Definitions

Similarly to the previous section, we first define the minimal y coordinate and a total order relationship now based on the y direction. We use this order to classify the processes in N sets $\{S_1, \dots, S_N\}$, which correspond to the N trails forming the final pattern. We then define the elementary distances between these sets and trails, and the y translation of a given trail.

- Similarly to Section 3.2, we define $y_{\min}$ as the smallest y coordinate occupied by a process. For a set of processes (resp. points) Q, let $y_{\min}(Q)$ be the smallest y coordinate occupied by a process (resp. point) of Q.
- The total order relationship defined in Section 3.2 gives priority to the x coordinate over the y coordinate. We now consider a total order relationship that gives priority to y over x. Let $(p_1, \dots, p_n)$ be the n processes ordered such that $p_1 < p_2 < \dots < p_n$, according to this order relationship. According to the definition of $\{T_1, \dots, T_N\}$ in Section 2.4, let S_1 be the $|T_1|$ first processes of the sequence, let S_2 be the $|T_2|$ following processes, and so forth until S_N.
- For two sets of processes $\{S, S'\} \subseteq \{S_1, \dots, S_N\}$, let $d(S, S') = \min_{(p,q)\in S\times S'} d(p, q)$. $\forall i \in \{1, \dots, N-1\}$, let $d_i = d(S_i, S_{i+1})$ (distance between sets) and $D_i = d(T_i, T_{i+1})$ (distance between trails).
- $\forall i \in \{1, \dots, N\}$ and for any value y_0, let $T'_i(y_0)$ be the trail obtained by a translation of T_i of distance $|y_0 - y_{\min}(T_i)|$ in the positive y direction.
- For a process p, let $S(p) \in \{S_1, \dots, S_N\}$ be the set such that $p \in S(p)$ (the set containing p).

4.2 CFPF Algorithm

The algorithm is described in Figure 2. We provide an informal description of the algorithm below.

Similarly to the CFTF algorithm of Section 3, the CFTF algorithm performs in 4 successive phases. These phases are defined so that the content of the sets $(S_1, \dots, S_N)$ always remains the same.

1. In Phase 1, the processes translate in the y direction until the smallest y coordinate of a process ($y_{\min}$) corresponds to the smallest y coordinate of the set of final positions ($y_{\min}(S)$).
2. In Phase 2, the sets $(S_1, \dots, S_N)$ scatter in the y direction until we have $d_i \geq \max(D, D_i)$ for each i. $d_i \geq D$ ensures that the processes can form the desired trails without collision in the next phase. $d_i \geq D_i$ ensures that the y distance between the sets is at least the desired distance D_i. Thus, the resulting trails only have to reduce this distance in Phase 4 to obtain the desired pattern.

CFPF Algorithm

At each step, every process p executes the following algorithm, which is divided in 4 phases. The conditions of the 4 phases are defined to be mutually exclusive.

Predicates

- P_1: $y_{\min} = y_{\min}(S_1)$
- P_2: $\forall i \in \{1, \ldots, N-1\}$, $d_i \geq \max(D, D_i)$.
- P_3: $\forall i \in \{1, \ldots, N\}$, there exists y_i such that $S_i = T'_i(y_i)$.
- P_4: The n processes occupy the n desired positions.

Phase 1: Translation

Condition: $\neg P_1$

Action: If $|y_{\min} - y_0| < \epsilon$, move with vector $[0, y_0 - y_{\min}]$. Otherwise:
- If $y_{\min} > y_0$, move with vector $[0, -\epsilon]$.
- If $y_{\min} < y_0$, move with vector $[0, \epsilon]$.

Phase 2: Scattering

Condition: $P_1 \wedge \neg P_2 \wedge \neg P_3$

Action: Let i be the smallest integer such that $d_i < \max(D, D_i)$.
Then, if $S(p)$ belongs to $\{S_{i+1}, S_{i+2}, \ldots, S_N\}$, move with vector $[0, \epsilon]$.

Phase 3: Trail formation

Condition: $P_1 \wedge P_2 \wedge \neg P_3$

Action: Let i be such that $S(p) = S_i$. Let $y_i = y_{\min}(S_i)$. Then, execute the CFTF algorithm of Section 3 to form the trail $T'_i(y_i)$ with the other processes of S_i.

Phase 4: Gathering

Condition: $P_1 \wedge P_3 \wedge \neg P_4$

Action: Let i be the smallest integer such that $d_i > D_i$. Let $e = d_i - D$.
Then, if $S(p)$ belongs to $\{S_{i+1}, S_{i+2}, \ldots, S_N\}$:
- If $e \leq \epsilon$, move with vector $[0, -e]$.
- If $e > \epsilon$, move with vector $[0, -\epsilon]$.

Figure 2 CFPF Algorithm.

3. In Phase 3, the processes of each set S_i use the CFTF algorithm of Section 3 to form a trail similar to T_i, but translated in the y direction.
4. In Phase 4, the sets $(S_1, \ldots, S_N)$ gather in the y direction until we have $d_i = D_i$ for each i. Then, the processes form the N desired trails, and thus the desired pattern.

Note that the CFTF algorithm does not increase (resp. decrease) the maximal (resp. minimal) y coordinate occupied by a process of the trail. Thus, the condition of Phase 3 always remains true during the execution of the CFTF subroutine.

4.3 Time complexity

▶ **Theorem 4.** *The n processes occupy the n desired positions in $O(R)$ time, and the condition on the minimal distance is always respected.*

Proof. The proof is in 4 steps.

1. Suppose $\neg P_1$. The distance of translation in Phase 1 is at most R. Thus, P_1 is true after a time $T_1 \leq \lceil R/\epsilon \rceil = O(R)$. As the translation is the same for all processes, the safety condition is respected.
2. Suppose $P_1 \wedge \neg P_2 \wedge \neg P_3$. Let i be the integer described in Phase 2. Then, we have $d_i \geq \max(D, D_i)$ in at most $\lceil \max(D, D_i)/\epsilon \rceil$ time steps. Thus, $P_1 \wedge P_2 \wedge \neg P_3$ is true after a time $T_2 \leq \Sigma_{i=1}^{i=N-1} \lceil \max(D, D_i)/\epsilon \rceil \leq \Sigma_{i=1}^{i=N-1} \lceil D/\epsilon \rceil + \Sigma_{i=1}^{i=N-1} \lceil D_i/\epsilon \rceil \leq N(1 + D/\epsilon) + (N + \Sigma_{i=1}^{i=N-1} D_i/\epsilon) \leq (2 + D/\epsilon)N + R/\epsilon$. According to Section 2.4, N is the smallest integer such that $y_1 + ND/2 > y_2$. Thus, $y_1 + (N-1)D/2 \leq y_2$, $(N-1)D/2 \leq y_2 - y_1 \leq R$, and $N \leq 1 + 2R/D$. Then, $T_2 \leq 2 + D/\epsilon + (4/D + 3/\epsilon)R = O(R)$. As the moves of Phase 2 only increase the distance between processes, the safety condition is respected.
3. Suppose $P_1 \wedge P_2 \wedge \neg P_3$. Let $i \in \{1, \ldots, N\}$. The moves of Phase 1 and 2 do not increase the distance between the processes of S_i. Therefore, all the processes of S_i are still contained in a circle of diameter R. Thus, as the processes of S_i execute the CFTF algorithm in Phase 3, according to Theorem 3, $P_1 \wedge P_3 \wedge \neg P_4$ is true after a time $T_3 = O(R)$. Let Y_i (resp. Y_i') be the smallest (resp. largest) y coordinate occupied by a process of S_i. During Phase 3, according to the CFTF algorithm, Y_i does not increase and Y_i' does not decrease. Thus, during Phase 3, $\forall i \in \{1, \ldots, N-1\}$, we have $d_i \geq D$, and the safety condition is respected.
4. Suppose $P_1 \wedge P_3 \wedge \neg P_4$. $\forall i \in \{1, \ldots, N-1\}$, Phase 2 increases d_i of at most $\max(D, D_i) + \epsilon$. Let Δ_i be the value of d_i at the beginning of Phase 4. Then, $\Sigma_{i=1}^{i=N-1} \Delta_i \leq R + \Sigma_{i=1}^{i=N-1} (\max(D, D_i) + \epsilon) \leq R + N(D + \epsilon) + \Sigma_{i=1}^{i=N-1} D_i \leq D + \epsilon + 2(2 + \epsilon/D)R$. Let i be the integer described in Phase 4. Then, we have $d_i = D_i$ in at most $\lceil \Delta_i/\epsilon \rceil \leq 1 + \Delta_i/\epsilon$ time steps. Thus, P_4 is true after a time $T_4 \leq \Sigma_{i=1}^{i=N-1} (1 + \Delta_i/\epsilon) \leq N + (\Sigma_{i=1}^{i=N-1} \Delta_i)/\epsilon \leq N + D/\epsilon + 1 + 2(2/\epsilon + 1/D)R = O(R)$. As the moves of Phase 4 do not allow to have $d_i < D_i$, the safety condition is respected.

Once P_4 is true, the processes do not move, and thus the liveness condition is satisfied. The execution time is at most $T_1 + T_2 + T_3 + T_4 = O(R)$. ◀

5 Conclusion

We gave and proved the first algorithm for pattern formation in the plane that always preserves a minimal distance between processes. We showed that the time complexity of this problem depends linearly on the diameter of the set of initial and final positions, and that our algorithm matches this bound.

A difficult open problem would be to study collision-free pattern formation in the presence of defective or malicious processes, that move independently of the algorithm and perturb the pattern formation. Also, an important step towards practical applications would be to study the impact of small errors and imprecision in the moves of processes and in their visibility mechanism. We also adopted a model where the processes have a global visibility of the swarm, and a challenging open problem would be to consider limited visibility.

References

1 Chrysovalandis Agathangelou, Chryssis Georgiou, and Marios Mavronicolas. A distributed algorithm for gathering many fat mobile robots in the plane. In *ACM Symposium on Principles of Distributed Computing, PODC'13, Montreal, QC, Canada, July 22-24, 2013*, pages 250–259, 2013. doi:10.1145/2484239.2484266.

2 Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

3 Leoncio Briones, Paul Bustamante, and Miguel Serna. Wall-climbing robot for inspection in nuclear power plants. In *Robotics and Automation*, pages 1409–1414. IEEE, 1994.

4 Sruti Gan Chaudhuri and Krishnendu Mukhopadhyaya. Gathering asynchronous transparent fat robots. In *Distributed Computing and Internet Technology*, pages 170–175. Springer, 2010.

5 Jurek Czyzowicz, Leszek Gasieniec, and Andrzej Pelc. Gathering few fat mobile robots in the plane. *Theoretical Computer Science*, 410(6):481–499, 2009.

6 Jurek Czyzowicz, Evangelos Kranakis, and Eduardo Pacheco. Localization for a system of colliding robots. In *Automata, Languages, and Programming*, pages 508–519. Springer, 2013.

7 Suparno Datta, Ayan Dutta, Sruti Gan Chaudhuri, and Krishnendu Mukhopadhyaya. Circle formation by asynchronous transparent fat robots. In *Distributed Computing and Internet Technology, 9th International Conference, ICDCIT 2013, Bhubaneswar, India, February 5-8, 2013. Proceedings*, pages 195–207, 2013. doi:10.1007/978-3-642-36071-8_15.

8 Xavier Défago and Samia Souissi. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theoretical Computer Science*, 396(1):97–112, 2008.

9 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, Thim Strothmann, and Shimrit Tzur-David. Infinite object coating in the amoebot model. *CoRR*, abs/1411.2356, 2014. URL: http://arxiv.org/abs/1411.2356.

10 Yoann Dieudonné and Franck Petit. Scatter of robots. *Parallel Processing Letters*, 19(01):175–184, 2009.

11 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

12 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by oblivious mobile robots. *Synthesis Lectures on Distributed Computing Theory*, 3(2):1–185, 2012.

13 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(1):412–447, 2008.

14 Nao Fujinaga, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern formation through optimum matching by oblivious CORDA robots. In *Principles of Distributed Systems – 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 1–15, 2010. doi:10.1007/978-3-642-17653-1_1.

15 Nao Fujinaga, Yukiko Yamauchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM J. Comput.*, 44(3):740–785, 2015. doi:10.1137/140958682.

16 Martin T. Hagan, Howard B. Demuth, Mark H. Beale, et al. *Neural network design*. Pws Pub. Boston, 1996.

17 Anthony Honorat, Maria Potop-Butucaru, and Sébastien Tixeuil. Gathering fat mobile robots with slim omnidirectional cameras. *Theoretical Computer Science*, 557:1–27, 2014.

18 Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The international journal of robotics research*, 5(1):90–98, 1986.

19 Sylvain Martel, Mahmood Mohammadi, Ouajdi Felfoul, Zhao Lu, and Pierre Pouponneau. Flagellated magnetotactic bacteria as controlled mri-trackable propulsion and steering systems for medical nanorobots operating in the human microvasculature. *The International journal of robotics research*, 28(4):571–582, 2009.

20 Michael S. Mooring and Benjamin L. Hart. Animal grouping for protection from parasites: selfish herd and encounter-dilution effects. *Behaviour*, 123(3):173–193, 1992.

21 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

22 Stephen Wolfram et al. *Theory and applications of cellular automata*, volume 1. World Scientific Singapore, 1986.

23 Yukiko Yamauchi, Taichi Uehara, Shuji Kijima, and Masafumi Yamashita. Plane formation by synchronous mobile robots in the three dimensional euclidean space. In *Distributed Computing – 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 92–106, 2015. doi:10.1007/978-3-662-48653-5_7.

Predicate Detection for Parallel Computations with Locking Constraints

Yen-Jung Chang[1] and Vijay K. Garg[2]

1 **Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA**
cyenjung@utexas.edu

2 **Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA**
garg@ece.utexas.edu

Abstract

The happened-before model (or the poset model) has been widely used for modeling the computations (execution traces) of parallel programs and detecting predicates (user-specified conditions). This model captures potential causality as well as locking constraints among the executed events of computations using Lamport's happened-before relation. The detection of a predicate in a computation is performed by checking if the predicate could become true in any reachable global state of the computation. In this paper, we argue that locking constraints are fundamentally different from potential causality. Hence, a poset is not an appropriate model for debugging purposes when the computations contain locking constraints. We present a model called Locking Poset, or a Loset, that generalizes the poset model for locking constraints. Just as a poset captures possibly an exponential number of total orders, a loset captures possibly an exponential number of posets. Therefore, detecting a predicate in a loset is equivalent to detecting the predicate in all corresponding posets. Since determining if a global state is reachable in a computation is a fundamental problem for detecting predicates, this paper first studies the reachability problem in the loset model. We show that the problem is NP-complete. Afterwards, we introduce a subset of reachable global states called lock-free feasible global states such that we can check whether a global state is lock-free feasible in polynomial time. Moreover, we show that lock-free feasible global states can act as "reset" points for reachability and be used to drastically reduce the time for determining the reachability of other global states. We also introduce strongly feasible global states that contain all reachable global states and show that the strong feasibility of a global state can be checked in polynomial time. We show that strong feasibility provides an effective approximation of reachability for many practical applications.

1998 ACM Subject Classification D.2.4 [Software/Program Verification] Validation

Keywords and phrases predicate detection, parallel computations, reachable global states, locking constraints, happened-before relation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.17

1 Introduction

One of the fundamental problems in debugging or runtime verification of a parallel program is to check if a *predicate* (user-specified condition) could become true in any global state that can be reached by the program. This problem is challenging because different runs of the program may reach different sets of global states due to the nondeterministic thread scheduling even for the same user input. In this paper, we propose a new model of parallel

Thread t_1	Thread t_2
$a1$:acquireLock(l)	$b1$:acquireLock(l)
$a2$: openFile(f)	$b2$: openFile(f)
$a3$:releaseLock(l)	$b3$: closeFile(f)
$a4$:closeFile(f)	$b4$:releaseLock(l)

Figure 1 A program which has two threads that might open the file `f` at the same time. The possible posets of its execution are shown in Fig. 2.

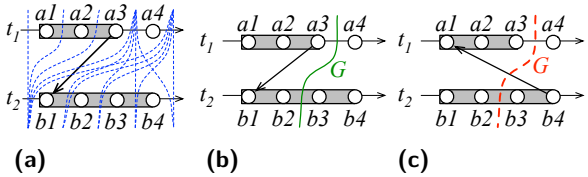


Figure 2 (a) The dashed lines are consistent global states. (b) Φ only becomes true in the global state G and it can be correctly detected because G is consistent. (c) In this poset, G is inconsistent and thus Φ cannot be detected.

computation that captures the reachable global states on multiple thread schedules and thus enables efficient predicate detection.

As an example of predicate detection, suppose that the condition Φ: *file f is opened by two threads at the same time*, is a potential bug of the parallel program shown in Fig. 1. We would like to know if the program can possibly reach a global state where Φ is true, i.e., to detect *possibly* Φ. One popular debugging method is to run the program and collect a totally ordered sequence of events. Suppose that the sequence recorded is $\sigma = a1, a2, a3, a4, b1, b2, b3, b4$. In this total order, Φ does not become true. However, the predicate is indeed possible if the sequence of events starts with the prefix $(a1, a2, a3, b1, b2)$. Hence, the only way to detect possibly Φ is to perform multiple executions and hope that one of them produces a total order that makes the predicate true [25, 31].

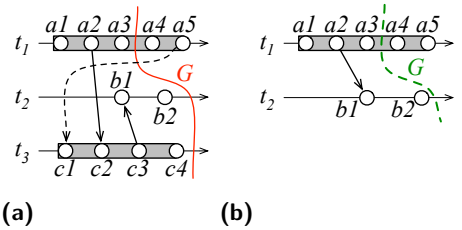
To alleviate this issue, the computation (the execution trace) of a parallel program is usually modeled as a partially ordered set (poset) of events, ordered by Lamport's happened-before relation (denoted by $\rightarrow$) [21]. In this poset, the events that are executed by a single thread are totally ordered and the events across threads are ordered based on their causality. Usually, the synchronization due to locks is also modeled with the happened-before relation. Specifically, the release of a lock is ordered before its subsequent acquisition [10, 22, 3, 5].

By modeling the computation as a poset, we are able to *predictively* detect the predicate if it becomes true in any *consistent global state* of the poset. In the poset model, a global state G is consistent iff for events e, f:$(e \rightarrow f) \wedge (f \in G) \Rightarrow (e \in G)$. Moreover, consistent global states are considered reachable because for every consistent global state there always exists at least one sequence of events that leads the program to reach that global state [1]. Hence, detecting a predicate on one poset is equivalent to detecting the predicate on multiple sequences of events. In addition, if we do not know the nature of the predicate, then predicate detection is usually done by enumerating all consistent global states of the poset and checking if any one of them satisfies the predicate [6, 18, 5, 3].

For the program in Fig. 1, the executions that produce σ and any total order with the prefix $(a1, a2, a3, b1, b2)$ are modeled as the same poset shown in Fig. 2a, in which the dashed lines are consistent global states of the poset and each of which contains all the events on its left. Fig. 2b shows the only global state G where the predicate Φ becomes true. Since G is consistent, Φ would be successfully detected when G is enumerated. However, we still have not solved the problem of predicate detection for all thread schedules. Suppose that thread t_2 obtains the lock before t_1 during the execution. Then, we put a happened-before order from $b4$ to $a1$ instead of $a3$ to $b1$ as shown in Fig. 2c. In this poset, G is inconsistent and will not be enumerated. Consequently, a purely poset based predicate detection algorithm will miss the predicate under a different locking schedule.

```
Thread t1                Thread t2
-----------              -----------
a1:acquireLock(l)        b1:recMsg(t3,&m)
a2: notify(l)            b2:openFile(f)
a3: openFile(f)
a4: closeFile(f)
a5:releaseLock(l)        Thread t3
                         -----------
                         c1:acquireLock(l)
                         c2: waitUntilNotified(l)
                         c3: sendMsg(t2,m)
                         c4:releaseLock(l)
```

Figure 3 A program which has three threads but the file `f` can only be opened by one thread at a time.



(a) **(b)**

Figure 4 (a) The global state G, where Φ is true, is indeed unreachable because of the implicit order (the dashed arrow) between the two critical sections. (b) The local view that contains only two of the threads, where G is mistakenly considered reachable.

An alternative approach removes the happened-before (HB) relation due to lock synchronization and determines the reachability of a global state using the techniques of lockset and acquisition history instead of the HB consistency of the global state [28, 19, 20, 29, 26]. However, these techniques only consider predicates that involve two threads, i.e., data races and atomicity violations. If the computation contains more than two threads, the detection is performed on a local view that consists of only two threads at a time. Hence, they can induce false positives because of the lack of the global view. Consider the program in Fig. 3, which has three threads. Because of the conditional synchronization (e.g., Java's `notify()` and `wait()`) between events $a2$ and $c2$, thread t_1 will obtain the lock l before t_3; otherwise, t_3 will be forced to release the lock. Thus, we get a computation as shown in Fig. 4a. Although the order $a5 \rightarrow c1$ is not explicitly captured in the computation, it is always implicitly induced during the execution of the program. Thus, the global state G, where Φ is true, is indeed unreachable. If we try to detect Φ in a local view that contains only two threads (see Fig. 4b), then G could be mistakenly considered reachable and result in a false-positive.

To deal with the co-existence of locks and the happened-before (HB) relation, one commonly used method is to convert mutual exclusion constraints and the HB relation to the constraints for SAT/SMT solvers [33, 34, 17]. When a global state that satisfies Φ is found, the solver is invoked in order to determine whether that global state is reachable in the computation. If it is reachable, then possibly Φ is detected. Although this method is applicable for detecting predicates that involve the global view of the system, these solvers may take exponential time in the worst case.

Since determining the reachability of a global state is a fundamental problem for the technique of predicate detection, our focus in this paper is on methods that take polynomial time for determining the reachability. We first introduce a model, named *Loset* (**Lo**cking po**set**), which is a generalization of the poset model. A Loset is a Poset augmented with the notion of locking intervals. In a loset, a lock synchronization is not modeled using the HB relation. Instead, the intervals of events that are executed under one or more locks are modeled separately. If two intervals $I1$ and $I2$ are executed under the same lock, then it is understood that events in $I1$ and $I2$ cannot be interleaved but can happen in either order. Since there can be an exponential number of different locking schedules, a loset in effect would model an exponential number of posets. Thus, a loset allows us to detect possibility of violation of invariants which would not be possible to detect using a single conventional poset. Moreover, our technique does not depend on the nature of the predicate. Thus, it can be used for detecting the predicate whose nature is unknown and requires the global view of the system.

Afterwards, we study the complexity of reachability problem in a loset: Given a loset $\mathcal{L}$ and a global state G, the reachability problem asks if there exists a sequence of events that leads the program to reach G in $\mathcal{L}$. The reachability problem is trivial for a poset: G is reachable iff G is consistent [1]. However, we show that the reachability problem for a loset is NP-complete. Our proof uses the NP-completeness of the predicate control problem shown in [30].

To cope with the NP-completeness, we introduce a subset of reachable global states called *lock-free feasible global states* such that we can efficiently check whether a global state is lock-free feasible in polynomial time. In this paper, a global state is lock-free if it does not hold any lock. We show that the set of reachable lock-free feasible global states forms a finite distributive lattice under the usual less than relation $<$ of global states. With the property of distributive lattice, we show that the reachability of a global state G can be determined using only a subset $(F \backslash G)$ of events, where F is the greatest lock-free feasible global state such that $F \leq G$. Thus, lock-free feasible states act as "reset" points for reachability and can be used to drastically reduce the time for checking reachability, by limiting the calculation in a subcomputation rather than the entire computation.

We also introduce *strongly feasible global states* that contain all reachable global states such that checking whether a global state is strongly feasible for a loset can be done efficiently. For many practical applications, strongly feasible global states provide an effective approximation of reachability: We show that the set of strongly feasible global states is identical to the set of reachable global states for computations with two threads. Moreover, our experiments show that the gap between strong feasibility and reachability seldom exists in practice. We give a method to enumerate the strongly feasible global states of a loset. In comparison with two naive but accurate enumeration algorithms, which enumerate only reachable global states, our enumeration method shows that the strongly feasible property accurately models the reachable global states for all 11 benchmark programs while using only 15–40% of their runtime.

We note here that our techniques are orthogonal to the techniques using SAT/SMT solvers. Given a trace of a computation, instead of calculating the reachability of a global state G from the initial global state, we only need to compute if G is reachable from the greatest lock-free feasible global state that precedes G. Moreover, we only need to calculate the reachability with a SAT/SMT solver if G is strongly feasible.

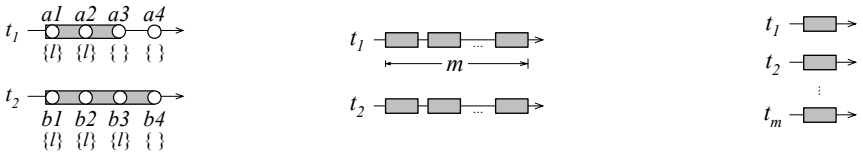
The rest of the paper is organized as follows. Section 2 presents the loset model. Section 3 and 4 introduce the sets of lock-free feasible and strongly feasible global states. Section 5 discusses the reachability of various classes of global states in a loset. Section 6 shows the experimental results. Finally, Section 7 concludes this paper.

2 Loset Model of a Computation

A computation (i.e., an execution trace of a parallel program) is modeled as a *Loset* (**Lo**cking Po**set**) of events as defined next.

▶ **Definition 1** (Loset). A loset $\mathcal{L}$ is a five-tuple $\mathcal{L} = (E, \rightarrow, n, L, \mathcal{I})$ where:

- E: is a set of events,
- $\rightarrow$: is an irreflexive transitive binary relation on E,
- n: is the number of threads,
- L: is the number of locks,
- $\mathcal{I}$: is a set of locking intervals.



(a) (b) (c)

Figure 5 (a) The loset that is equivalent to the two posets in Fig. 2b and 2c. The gray boxes are the critical sections created by the same lock. The curly braces show the locks that are held at each event. (b) A loset that is equivalent to C_m^{2m} posets. (c) A loset that is equivalent to $m!$ posets.

The $\rightarrow$ relation represents the potential causality between events, i.e., $e \rightarrow f$ means that the event e may directly or transitively cause the event f. For distributed systems, it corresponds to the Lamport's happened-before (HB) relation [21]. In concurrent systems, we may have additional order constraints due to the *fork-join* events of threads and the *wait-notification* events of conditional synchronization [10, 22, 3, 5]. In this paper, we maintain the $\rightarrow$ relation using vector clocks [9, 23]. The set E of events is partitioned into n sequences $E_1, E_2, \cdots, E_n$ such that each E_i represents a thread. For all distinct events $e, f \in E_i : (e \rightarrow f) \vee (f \rightarrow e)$. For convenience, we define the process order relation (denoted by $\prec$) such that $e \prec f$ means $e \rightarrow f$ in some E_i. A locking interval $I \in \mathcal{I}$ is a four-tuple $I = (t, l, acq, rel)$ such that $t \in \{1..n\}, l \in \{1..L\}$, $(acq, rel \in E_t)$, and $acq \prec rel$. An interval indicates that the thread $I.t$ acquired the lock $I.l$ at event $I.acq$ and released it at $I.rel$.

Note that the objective of the $\rightarrow$ relation is to capture the causality of events but not the real-time locking order between the acquisition and release events of locks. Therefore, the locking intervals for the same lock are totally ordered in a poset but not in a loset. Formally,

▶ **Definition 2** (Valid Poset of a Loset). A poset $P = (E, \rightarrow_P)$ is a *valid poset* of a loset $\mathcal{L} = (E, \rightarrow, n, L, \mathcal{I})$ if $(\rightarrow \subseteq \rightarrow_P)$ and $\forall I, J \in \mathcal{I}$ such that $I.l = J.l$, we have $(I.rel \rightarrow_P J.acq) \vee (J.rel \rightarrow_P I.acq)$.

For instance, the two posets in Fig. 2b and Fig. 2c are the valid posets of the loset in Fig. 5a. In Fig. 5b, suppose that each thread contains m locking intervals for the same lock, then the loset is equivalent to C_m^{2m} valid posets because the m intervals of t_1 can be interleaved with those of t_2 in C_m^{2m} total orders. Similarly, the loset in Fig. 5c is equivalent to $m!$ valid posets. Fig. 7 shows a more complex loset. We now define global states and their reachability under the loset model.

2.1 Global States

A **global state** G is a subset of E such that $\forall e, f \in E : (f \in G) \wedge (e \prec f) \Rightarrow (e \in G)$. In Fig. 5a, the set $\{a1, a2, b1\}$ is a global state, but $\{a2, b1\}$ is not a global state because it contains event $a2$ but not $a1$ even though $a1 \prec a2$. A global state G can equivalently be identified by the number of events of each E_i in G. For example, the global state $\{a1, a2, b1\}$ is represented by the array $[2, 1]$. The symbol $G[i]$ denotes the maximal (latest) event of E_i in the global state G. The order $G \leq H$ between the two global states means $G[i] \preceq H[i]$ holds for any thread i.

A global state G is **consistent** iff $\forall e, f \in E : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G)$. A consistent global state preserves the $\rightarrow$ relation of the loset. Note that the initial global state $(G = \phi)$, and the final global state $(G = E)$ are always consistent. We define the set $\text{EL}(e)$ of **effective locks** for any event e, which are the locks being held by the thread that has executed e:

$$\text{EL}(e) = \{I.l \mid I.acq \preceq e \prec I.rel\}.$$

In Fig. 5a, the effective locks of the events in the computation are shown in curly brackets. We can now define the set of global states that respect the locking constraints. A global state G is (lock) **compatible** iff for any $i \neq j, G[i]$ and $G[j]$ are pairwise (lock) compatible, i.e., $\text{EL}(G[i]) \cap \text{EL}(G[j]) = \varnothing$. Finally, a global state is **feasible** iff it is consistent and compatible.

If a global state is not feasible then it violates either the consistency constraints or the locking constraints. Hence, only feasible global states are reachable from the initial global state. However, not all feasible global states are reachable. For example, the global state G in Fig. 4a is feasible but not reachable because of the implicit locking order induced by the conditional synchronization.

2.2 Reachable Global States and Runs

We first introduce a sequence of events called a run, $\mathcal{R}$, in which the total order between events is denoted by $\prec_{\mathcal{R}}$. The symbol $\delta(G, \mathcal{R})$ denotes the global state that is reached by executing the sequence $\mathcal{R}$ of events starting from the global state G. The symbol $\mathcal{R}^i$ denotes the prefix of $\mathcal{R}$ of length i. Since only feasible states are reachable, a *run* goes through only feasible global states. Formally, a sequence $\mathcal{R}$ of events is a **run** starting from G iff the global state $\delta(G, \mathcal{R}^i)$ is feasible for any i such that $0 \leq i \leq |\mathcal{R}|$.A global state G is **reachable** from the initial global state ϕ iff there exists a run $\mathcal{R}$ such that $\delta(\phi, \mathcal{R}) = G$. The reachability problem is defined as:

▶ **Definition 3** (Loset Reachability Problem)**.** Given a loset $\mathcal{L}$ and a global state G, is G a reachable global state of $\mathcal{L}$?

▶ **Theorem 4.** *The loset reachability problem is NP-complete.*

Proof. (Outline) In [30], the predicate control problem asks if there exists a control sequence, which is a total order among the critical sections for the same lock, such that the predicate Φ remains true after the control sequence is added to the computation. It was shown that the predicate control problem is NP-complete. The model defined in [30] is a special case of our loset model, where locking intervals do not overlap. It can be shown that there exists a control sequence that reaches the global state G without violating mutual exclusion iff the global state G is reachable in the loset. Therefore, the predicate control problem is a special case of the reachability problem of a loset. The details are available in [2]. ◀

In the following sections, we present two classes of global states — lock-free feasible global states and strongly feasible global states. A lock-free feasible global state is always reachable and a reachable global state is always strongly feasible. Thus, these two classes provide a lower and an upper bound on the set of reachable global states. Both of these classes can be checked efficiently (in polynomial time), whereas the reachability problem is NP-complete. Moreover, to check reachability of a global state G, it is sufficient to check its reachability from the greatest lock-free feasible global state that precedes G instead of checking it from the initial global state of the computation.

3 Lock-Free Feasible Global States

A *lock-free feasible global state* is a feasible global state that holds no lock. We show that given a reachable global state G of a loset, then any lock-free feasible global state $F \leq G$ is also reachable.

▶ **Theorem 5.** *Given a reachable global state G of a loset and a lock-free feasible global state $F \leq G$, there exists a run that reaches both F and G.*

Proof. Since G is reachable, there exists a run $\mathcal{R}$ such that $\delta(\phi, \mathcal{R}) = G$. Let the sequence $\mathcal{S}_1$ of events be $\mathcal{R} \uparrow F$, which is the projection of $\mathcal{R}$ that contains only the events in F, and let $\mathcal{S}_2 = \mathcal{R} \uparrow (G \backslash F)$. Let $\mathcal{S} = \mathcal{S}_1 \oplus \mathcal{S}_2$ ($\mathcal{S}_1$ concatenated with $\mathcal{S}_2$). We show that the sequence $\mathcal{S}$ of events is also a run, i.e., $\delta(\phi, \mathcal{S}^i)$ is feasible for any $\mathcal{S}^i$, which implies $\delta(\phi, \mathcal{S}_1) = F$ and $\delta(F, \mathcal{S}_2) = G$.

Claim 1. $\forall i : 0 \leq i \leq |\mathcal{S}| : \delta(\phi, \mathcal{S}^i)$ is consistent

We show the partial order $\rightarrow$ of the computation is preserved by the total order $\prec_{\mathcal{S}}$ in $\mathcal{S}$. For any two events, e and f, in $\mathcal{S}$ such that $e \prec_{\mathcal{S}} f$, we have

Case 1. $(e, f \in \mathcal{S}_1) \vee (e, f \in \mathcal{S}_2)$: The $\rightarrow$ relation between e and f is preserved in $\prec_{\mathcal{R}}$ because $\mathcal{R}$ is a run. Since $\mathcal{S}_1$ and $\mathcal{S}_2$ are projections of $\mathcal{R}$, the $\rightarrow$ relation is preserved in $\prec_{\mathcal{S}_1}$ and $\prec_{\mathcal{S}_2}$.

Case 2. $e \in \mathcal{S}_1, f \in \mathcal{S}_2$: If $e \rightarrow f$, the $\rightarrow$ relation is preserved by the concatenation $\mathcal{S}_1 \oplus \mathcal{S}_2$. The case $f \rightarrow e$ is not possible because F is consistent and $e \in F$ but $f \notin F$.

Claim 2. $\forall i : 0 \leq i \leq |\mathcal{S}_1| : \delta(\phi, \mathcal{S}_1^i)$ is compatible

Let the global state $V = \delta(\phi, \mathcal{S}_1^i)$. We show that

$$\forall s \neq t : \text{EL}(V[s]) \cap \text{EL}(V[t]) = \varnothing. \tag{1}$$

Let $\mathcal{R}^j$ be the shortest prefix of $\mathcal{R}$ such that $\mathcal{R}^j \uparrow F = \mathcal{S}_1^i$ and let $W = \delta(\phi, \mathcal{R}^j)$. Then, the following condition holds because $\mathcal{R}$ is a run:

$$\forall s \neq t : \text{EL}(W[s]) \cap \text{EL}(W[t]) = \varnothing. \tag{2}$$

Since $\mathcal{S}_1^i$ contains the same or fewer events than $\mathcal{R}^j$, we get $V \subseteq W$, which implies $V[t] \preceq W[t]$ for any thread t. We now consider the following two cases:

Case 1. $V[t] \prec W[t]$: Because $\mathcal{S}_1^i = \mathcal{R}^j \uparrow F$, this case holds only if $\mathcal{R}^j$ contains the events in $G \backslash F$ w.r.t. E_t, which implies that $\mathcal{S}_1^i$ contains all the events in F w.r.t. E_t. Thus, we get $V[t] = F[t] \prec W[t]$. Since F is lock-free, we get $\text{EL}(V[t]) = \varnothing \subseteq \text{EL}(W[t])$.

Case 2. $V[t] = W[t]$: In this case, we get $\text{EL}(V[t]) = \text{EL}(W[t])$.

From cases 1 and 2, $\text{EL}(V[t]) \subseteq \text{EL}(W[t])$ holds for any thread t. Then, from (2), (1) holds.

Claim 3. $\forall i : 0 \leq i \leq |\mathcal{S}_2| : \delta(F, \mathcal{S}_2^i)$ is compatible

Let the global state $V = \delta(F, \mathcal{S}_2^i)$. We show that

$$\forall s \neq t : \text{EL}(V[s]) \cap \text{EL}(V[t]) = \varnothing. \tag{3}$$

Let $\mathcal{R}^j$ be the shortest prefix of $\mathcal{R}$ such that $\mathcal{R}^j \uparrow (G \backslash F) = \mathcal{S}_2^i$ and $W = \delta(\phi, \mathcal{R}^j)$. Then, the following condition holds because $\mathcal{R}$ is a run:

$$\forall s \neq t : \text{EL}(W[s]) \cap \text{EL}(W[t]) = \varnothing. \tag{4}$$

Since V initially contains all the events in F and $\mathcal{S}_2^i$ contains the same events in $G \backslash F$ as $\mathcal{R}^j$, we get $W \subseteq V$, which implies that $W[t] \preceq V[t]$ holds for any thread t:

Case 1. $W[t] \prec V[t]$: Because $\mathcal{S}_2^i = \mathcal{R}^j \uparrow G \backslash F$, this case holds only if $\mathcal{R}^j$ contains only the events in F w.r.t. E_t, which implies that $\mathcal{S}_2^i$ does not contain any event of E_t. Thus, we get $W[t] \prec V[t] = F[t]$. Since F is lock-free, we get $\text{EL}(W[t]) \supseteq \text{EL}(V[t]) = \varnothing$.

Case 2. $W[t] = V[t]$: We get $\text{EL}(W[t]) = \text{EL}(V[t])$.

From the two cases, $\text{EL}(W[t]) \supseteq \text{EL}(V[t])$ holds for any thread t. Then, from (4), (3) holds.

From claims 1, 2, and 3, $\mathcal{S}$ is a run that reaches first F using the run $\mathcal{S}_1$ and then reaches G using the run $\mathcal{S}_2$. ◀

Since we use the loset model for analyzing the behavior of parallel programs, we are interested only in those losets that capture a possible execution from a real-world application, i.e., the reachability of the final global state of the computation is given by the execution of the program. Formally, a loset is **valid** iff its final global state E is reachable. An example of a loset, which is an artificial computation, that is not valid is available in [2]. A simple consequence of Theorem 5 is that whenever $\mathcal{L}$ is a valid loset, then every lock-free feasible global state of $\mathcal{L}$ is reachable.

▶ **Corollary 6.** *All lock-free feasible global states of a valid loset are reachable.*

Proof. The final global state of a valid loset is reachable. Therefore, from Theorem 5, we get that every lock-free feasible global state of that loset is reachable. ◀

The set of *reachable* lock-free feasible global states also satisfies the following nice property for all losets: (and not just valid losets).

▶ **Theorem 7.** *The set of reachable lock-free feasible global states of a loset $\mathcal{L}$ forms a distributive lattice.*

Proof. (Outline) For any two reachable lock-free feasible global states, G and H, let $M = (G \cap H)$ be their meet and $J = (G \cup H)$ be their join. We first show that M and J are lock-free and feasible. Then, from Theorem 5, M is reachable because $M \leq G$. To show their join J is reachable, we construct a sequence $\mathcal{S}_J$ of events such that $\mathcal{S}_J = \mathcal{R}_G \oplus \mathcal{R}_{MH}$, where R_G is a run reaches G and $\mathcal{R}_{MH}$ reaches H from M. Then, we show that $\mathcal{S}_J$ is also a run. The details are available in [2]. ◀

Theorem 7 has two important implications. First, since the set of reachable lock-free feasible global states forms a distributive lattice, we can concisely represent all lock-free feasible global states of a valid loset using the set of *join-irreducible* elements of the distributive lattice [7] and use slicing to study various sublattices, which reduces the time complexity of predicate detection to polynomial time for certain classes of predicates [13, 24]. Secondly, as shown next, we can reduce the search space to determine reachability of a feasible global state that is not lock-free. Given a global state G, we first find the greatest lock-free feasible global state $F \leq G$. On account of Theorem 7, F is well-defined whenever there exists any lock-free feasible global state that precedes G. Given G and F, the following theorem shows that the search for the reachability in a valid loset can be restricted to the events in $G \backslash F$.

▶ **Theorem 8.** *Given a global state G of a valid loset and the greatest lock-free feasible global state F such that $F \leq G$, the reachability of G can be determined by the events $G \backslash F$.*

Proof. From Theorem 5, F is reachable because the final global state E is reachable. Moreover, the run that reaches E of $\mathcal{L}$ can be reordered so that it first reaches F and then E. We consider the following two cases: (1) If G is reachable, then from Theorem 5 there exists a run $\mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2$, where $\mathcal{R}_1$ is a run that reaches F and $\mathcal{R}_2$ is a run that reaches G from F. (2) If G is unreachable, then there exists no run from F to G because F is reachable and lock-free. Hence, the existence of the run $\mathcal{R}_2$ depends only on the events $G \backslash F$. ◀

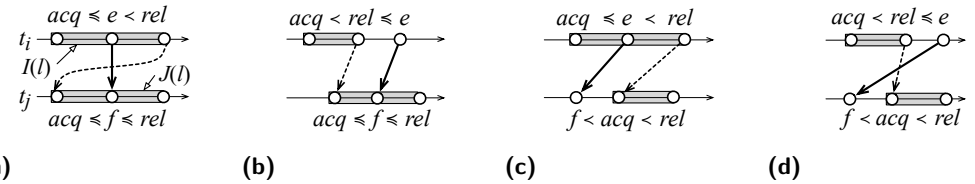


Figure 6 All possible cases of $I(l) \mapsto J(l)$ and the locking order $I(l).rel \rightarrow_L J(l).acq$ (shown in dashed lines).

4 Strongly Feasible Global States

In this section, we give an upper-approximation of reachability. We define the notion of *strong feasibility* based on the inferred causality due to the HB relation and locking constraints. Therefore, a reachable global state is always strongly feasible. Also, just as feasibility and lock-freedom can be evaluated in polynomial time, strong feasibility can be evaluated in polynomial time.

4.1 Locking Order

Even though real-time locking order is not modeled in a loset, some order between locks may be implied due to the HB orders between events and locking constraints (i.e., the events in different locking intervals of the same lock cannot be interleaved during the execution of the program). We next introduce the relation $\mapsto$ for capturing such implied ordering constraints.

The $\mapsto$ relation is defined between locking intervals of the same lock such that $I \mapsto J$ means the locking interval I has to start before J can finish:

▶ **Definition 9** (Relation $\mapsto$)**.** Let $I(l)$ and $J(l)$ be the locking intervals of the same lock l. $I(l) \mapsto J(l)$ iff there exist events e and f such that $(I(l).acq \preceq e) \wedge (e \rightarrow f) \wedge (f \preceq J(l).rel)$.

Fig. 6 shows all possible cases of $I(l) \mapsto J(l)$. Because of the locking constraint from the lock l, the event $I(l).rel$ has to be executed before $J(l).acq$. Hence, we define the *locking order* $\rightarrow_L$ as follows:

▶ **Definition 10** (Locking Order $\rightarrow_L$)**.** $(e \rightarrow_L f)$ iff there exists two locking intervals, $I(l)$ and $J(l)$, of the same lock l such that $(e = I(l).rel) \wedge (I(l) \mapsto J(l)) \wedge (f = J(l).acq)$.

If $I(l)$ and $J(l)$ belong to the same thread, then the $\rightarrow_L$ relation is implied by their process order. Therefore, we only consider the $\rightarrow_L$ relation across different threads. Fig. 6 shows the corresponding locking order of all possible cases of $I(l) \mapsto J(l)$ in the dashed lines. For convenience, the locking order $I(l).rel \rightarrow_L J(l).acq$ is simplified as $I(l) \rightarrow J(l)$ from now on.

In this paper, we assume for simplicity that the initial global state does not hold any lock. If it is not lock-free, then any interval $I(l)$ that is part of the initial global state is ordered (by locking constraints) before all other intervals with the same lock l.

4.2 Normalization of Losets

Since the combination of happened-before orders and locking constraints may introduce additional order constraints $\rightarrow_L$ during execution, it is easier to analyze a loset that satisfies $\forall e, f : e \rightarrow_L f \Rightarrow e \rightarrow f$. Thus locking order leads us to the following definition:

▶ **Definition 11** (Normal Loset)**.** A loset $\mathcal{L} = (E, \rightarrow, n, L, \mathcal{I})$ is normal if $\forall e, f \in E : e \rightarrow_L f \Rightarrow e \rightarrow f$.

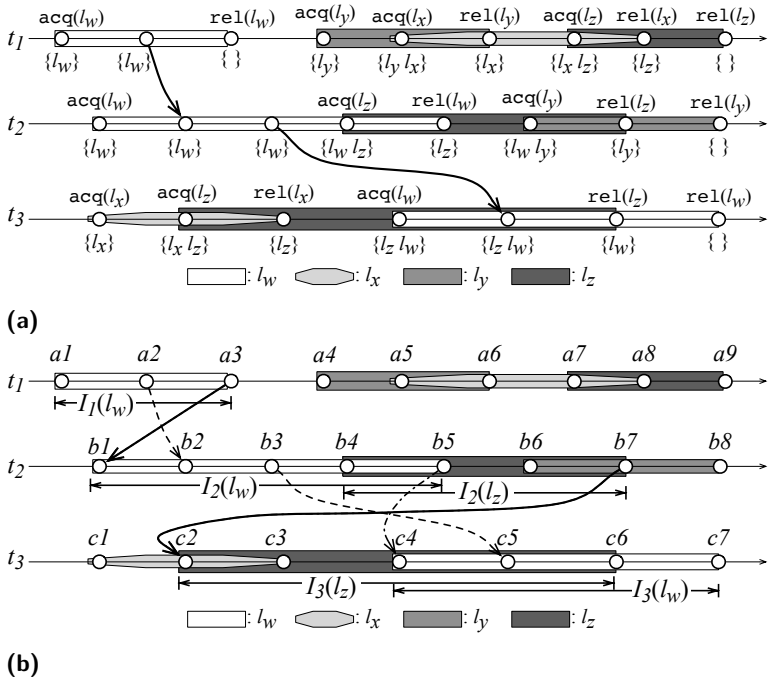


Figure 7 (a) An initial loset $\mathcal{L}$, which contains only the HB relation. (b) A normal loset $\mathcal{L}'$, where the locking orders (the solid arrows) are added to the original loset $\mathcal{L}$.

Fig. 7a shows a loset $\mathcal{L}$, which contains only the HB relation and four locks l_w, l_x, l_y, and l_z. The events `acq(l)` and `rel(l)` correspond to the operations `acquireLock(l)` and `releaseLock(l)` of the program, respectively. The solid arrows are direct HB orders between events. The boxes of different gray-levels are the locking intervals with different locks. The effective locks of events are shown in the curly brackets. Fig. 7b shows the corresponding normal loset $\mathcal{L}'$, which has locking orders added to $\mathcal{L}$. The dashed arrows in Fig. 7b are used to explain the procedure of normalization as shown next.

At first, the HB relation $a2 \to b2$ induces the relation $I_1(l_w) \mapsto I_2(l_w)$ and hence the locking order $a3 \to b1$. Therefore, the locking order $a3 \to b1$ is added to $\mathcal{L}$. Similarly, the HB relation $b3 \to c5$ induces the relation $I_2(l_w) \mapsto I_3(l_w)$ and hence the locking order $b5 \to c4$. Afterwards, the relation $b5 \to c4$ induces $I_2(l_z) \mapsto I_3(l_z)$ and hence the locking order $b7 \to c2$. The procedure continues until no new locking order is induced. Note that the transitive HB relation $a2 \to c5$ is not shown in Fig. 7b, which induces $I_1(l_w) \mapsto I_3(l_w)$ and hence the locking order $a3 \to c4$, because its corresponding locking order $a3 \to c4$ is transitively implied by other relations.

Algorithm 1 shows a procedure to normalize a loset $\mathcal{L}$. The algorithm takes as input the direct and transitive HB orders in the computation (i.e., $a2 \to b2$, $b3 \to c5$, and $a2 \to c5$ in Fig. 7a) and iteratively adds the locking orders to the computation by locating the cases of the $\mapsto$ relation in Fig. 6a, 6b, and 6c. The case of Fig. 6d is ruled out in Algorithm 1 because the locking order is transitively implied by $I(l) \mapsto J(l)$ and does not induce any new $\to$ relation. At line 9, if the addition of $I(l) \to J(l)$ induces any transitive relation, say $e \to f$, then $e \to f$ is also appended to the set $\mathcal{H}$ for checking if any new $\mapsto$ relation is induced.

We now discuss the time complexity of the normalization procedure.

▶ **Theorem 12.** *The time complexity of Algorithm 1 is* $O(n|E|^3L)$.

Algorithm 1 NormalizeLoset($\mathcal{L}$, $\mathcal{H}$)

Input: A loset $\mathcal{L}$ that contains only HB orders, which are added to the set $\mathcal{H}$ of seed relations.
Output: Returns false if a cycle in the $\rightarrow$ relation is detected; otherwise, the loset $\mathcal{L}$ is normalized.

1: **for** each seed order $e_i \rightarrow e_j$ in $\mathcal{H}$ **do** ▷ $\mathcal{H}$ initially contains all direct and transitive $\rightarrow$ relation.
2: **for** each $l \in EL(e_i) \cup EL(e_j)$ **do** ▷ Exclude the case of Fig. 6d.
3: Let $I(l)$ be the most recent locking interval for l s.t. $I(l).acq \preceq e_i$.
4: Let $J(l)$ be the first locking interval for l s.t. $e_j \preceq J(l).rel$.
5: **if** either $I(l)$ or $J(l)$ does not exist **then** continue ▷ None of the cases, Fig. 6a, 6b, or 6c, holds.
6: **if** the relation $I(l) \rightarrow J(l)$ completes a cycle **then return** false
7: **else**
8: Add $I(l) \rightarrow J(l)$ to the loset and to the set $\mathcal{H}$ ▷ $I(l) \rightarrow J(l)$ means $I(l).rel \rightarrow J(l).acq$.
9: Append new transitive relations due to $I(l) \rightarrow J(l)$ to $\mathcal{H}$
10: **end if**
11: **return** true

Proof. Line 1 executes at most $O(|E|^2)$ times because there are at most $O(|E|^2)$ pairs of the $\rightarrow$ relation in the computation. Line 2 executes at most L times. The procedures at lines 3 and 4 can be done in constant time by using lookup tables. Finally, the time complexity for detecting the cycle at line 6 and for locating the transitive relations at line 9 is $O(n|E|)$ by recomputing vector clocks after the addition of the relation $I(l) \rightarrow J(l)$ at line 8. ◀

We now show that the normal loset contains the same set of runs that reach the final global state as the original loset. We first define the runs $Runs(\mathcal{L})$ of a loset:

▶ **Definition 13** (Runs of a Loset)**.** Given any loset $\mathcal{L}$, the set $Runs(\mathcal{L}) = \{\mathcal{R} \mid \mathcal{R}$ is a run that reaches the final global state E of $\mathcal{L}$ from the initial global state $\phi\}$.

▶ **Theorem 14.** *Let $\mathcal{L}$ be a loset and $\mathcal{L}'$ be the corresponding normal loset, then $Runs(\mathcal{L}) = Runs(\mathcal{L}')$.*

Proof. (Sketch) We show that $Runs(\mathcal{L}') \subseteq Runs(\mathcal{L})$ and $Runs(\mathcal{L}) \subseteq Runs(\mathcal{L}')$. Since $\mathcal{L}'$ contains more constraints of the $\rightarrow$ relation, we get $Runs(\mathcal{L}') \subseteq Runs(\mathcal{L})$. On the other hand, it is easily shown that any run $\mathcal{R}$ in $Runs(\mathcal{L})$ is also a run of $Runs(\mathcal{L}')$ because the run $\mathcal{R}$ in $Runs(\mathcal{L}, E)$ does not violate any locking order constraint and therefore only goes through feasible states of $\mathcal{L}'$. ◀

4.3 Strong Feasibility

If a lock l is held by a thread i in the global state G, then any other thread, say, j, that acquired the lock l prior to G should have released it before thread i subsequently acquires it. We refer this implicit order due to G as *dynamic locking order*. Formally,

▶ **Definition 15** (Dynamic Locking Order $\rightarrow_L$)**.** $(e \rightarrow_L f)$ iff there exists two locking intervals, $I(l)$ and $J(l)$, of the same lock l such that $\big((e \in E_i) \wedge (e = I(l).rel \preceq G[i])\big) \wedge \big((f \in E_j) \wedge (f = J(l).acq \preceq G[j] \prec J(l).rel)\big)$.

The dynamic locking orders due to G can be added to $\mathcal{H}$ and then be normalized in order to estimate the reachability of G. We now define *strong feasibility* of a global state as follows:

▶ **Definition 16** (Strong Feasibility)**.** A feasible global state G is strongly feasible iff the normalization of the loset due to G does not induce any cycle in the $\rightarrow$ relation.

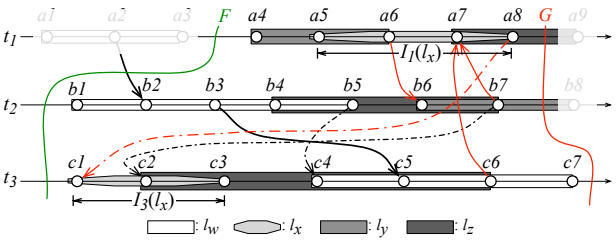


Figure 8 The feasible global state G is unreachable because the locking order completes a cycle in the $\rightarrow$ relation.

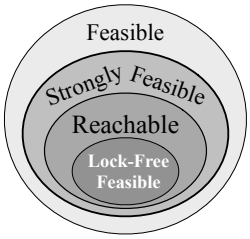


Figure 9 The relationship among various classes of global states in a valid loset.

We use the feasible global state $G = [8, 7, 7]$ in Fig. 8 to show the calculation of strong feasibility:

Step 1: From Theorem 8, this calculation can be bounded between G and the greatest lock-free feasible global state F that precedes G, i.e., the grayed out events in Fig. 8 are excluded.

Step 2: Since the lock l_z is currently held by the thread t_1, so we get the dynamic locking orders $c6 \rightarrow a7$ and $b7 \rightarrow a7$. Similarly, the lock l_y is currently held by the thread t_2, we get $a6 \rightarrow b6$.

Step 3: The HB orders of the sub-loset along with dynamic locking orders are added to the set $\mathcal{H}$ for normalization. From $b3 \rightarrow c5$, we get $b5 \rightarrow c4$ and then $b7 \rightarrow c2$. Then, the transitive relation $a6 \rightarrow c2$ establishes the relation $I_1(l_x) \mapsto I_3(l_x)$ and hence the locking order $a8 \rightarrow c1$. Consequently, a cycle in the $\rightarrow$ relation is induced: $a8 \rightarrow c1 \rightarrow c6 \rightarrow a7 \rightarrow a8$. Thus, G is not strongly feasible.

▶ **Theorem 17.** *The time complexity for calculating the strong feasibility of a global state is* $O(n|E|^3L)$.

Proof. In step 1, the lock-free feasible global state F can be identified using the detection algorithm for conjunctive predicate [14] starting from G in a backward fashion, which takes at most $O(|E|)$ time. In step 2, we can locate the dynamic locking orders due to G by pairwise processing the maximal events of G for each lock, which takes $O(n^2L)$ time. In step 3, the normalization takes at most $O(n|E|^3L)$ time using Algorithm 1. ◀

5 Relationship Among Various Classes of Global States

Fig. 9 shows the relationship among different sets of global states in a valid loset, whose final global state is reachable. Corollary 6 shows that all lock-free feasible global states are reachable and hence they are a subset of reachable global states. The set of strongly feasible global states is a superset of reachable global states: (1) Every reachable global state is strongly feasible because the normalization of a loset does not remove any run that reaches G, which can be shown by replacing E and $\mathcal{L}$ of Theorem 14 with G and the sub-loset from Theorem 8, respectively. Moreover, a reachable global state does not contain any cycle in $\rightarrow$ relation. (2) A strongly feasible global state may be unreachable; an example is shown in [2].

Strong feasibility is still useful in practice; we now show that reachability equals to strong feasibility in any loset with two threads:

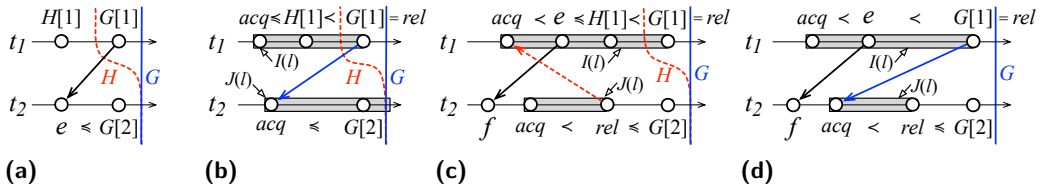


Figure 10 (a) Case 1: $H = G - G[1]$ is inconsistent. (b) Case 2: H is incompatible. (c) Case 3: H induces a cycle in the $\rightarrow$ relation and either $(f \preceq acq)$ or $(acq \preceq f)$ holds. (d) Case 3: The cycle in c implies $G[1] \rightarrow G[2]$.

▶ **Theorem 18.** *In a loset $\mathcal{L}$ with two threads, a global state is reachable iff it is strongly feasible.*

Proof. It is sufficient to show that any strongly feasible global state G of a loset with two threads is always reachable. We show this by induction on the size of G. When $|G| = 0$, G is the initial global state and therefore reachable. Now consider any G such that $|G| > 0$. We will show that there exists a maximal event e in G such that $G - \{e\}$ is also strongly feasible. By the induction hypothesis, we can assume that $G - \{e\}$ is reachable and therefore G is reachable.

We now show that there does not exist a strongly feasible global state G such that removing any of its maximal event results in a global state that is not strongly feasible. Let $H = G - G[1]$ and $F = G - G[2]$. Without loss of generality, we show that if H is not strongly feasible, then $G[1] \rightarrow G[2]$. We consider the following three cases:

Case 1. *H is not consistent*: It is obvious that $G[1] \rightarrow G[2]$. (See Fig. 10a.)

Case 2. *H is not compatible*: An example loset is shown in Fig. 10b. If H is not compatible, then there exists one lock $l \in \text{EL}(H[1]) \cap \text{EL}(G[2])$. Let $I(l)$ and $J(l)$ be the two intervals for the lock l such that $I(l).acq \preceq H[1] \prec I(l).rel$ and $J(l).acq \preceq G[2] \prec J(l).rel$. Since G is compatible (i.e., $\text{EL}(G[1]) \cap \text{EL}(G[2]) = \varnothing$), we get $G[1] = I(l).rel$. Consequently, the locking order $I(l).rel \rightarrow_L J(l).acq$ is induced in G and hence $G[1] \rightarrow G[2]$.

Case 3. *H contains a cycle in the $\rightarrow$ relation*: Fig. 10c shows an example loset. Since G is strongly feasible, the cycle must be completed by a locking order that is induced by H. Suppose that the locking order is induced because of the lock l, then the following conditions hold:

1. Since the locking order only exits in H, there exists an interval $I(l)$ such that $H[1] \prec I(l).rel = G[1]$.
2. There exists an interval $J(l)$ such that $J(l).rel \preceq G[2]$. Thus, the locking order $J(l).rel \rightarrow_L I(l).acq$ can be induced in H but not G.

In order to complete the cycle, there exists a relation $e \rightarrow f$ in H such that $I(l).acq \prec e \preceq H[1]$ and $f \prec J(l).rel$. Since the computation has only two threads, any locking order due to H must point toward the events that occur on t_1. Hence, the relation $e \rightarrow f$ is either an existing HB relation of the computation or a locking order that is induced by $G[2]$. In either case, $e \rightarrow f$ also exists in G. Then, $e \rightarrow f$ would induce the relation $I(l) \mapsto J(l)$ in G (see Fig. 10d) and hence the locking order $G[1] \rightarrow_L J(l).acq$, which implies $G[1] \rightarrow G[2]$.

Table 1 The information of benchmarks and runtimes (sec.) of each enumeration approach.

Benchmark	n	#events	#GS	Runtimes			n	#events	#GS	Runtimes		
				BFS	DFS	Ours				BFS	DFS	Ours
bank	7	91	664,325	0.99	3.20	0.09	9	121	53,808,433	350.27	o.o.m.	4.47
arraylist1	12	56	354,293	0.57	1.06	0.07	16	76	28,697,813	175.80	o.o.m.	1.66
arraylist2	7	103	3,045,808	4.48	30.28	0.22	8	118	25,740,144	104.81	o.o.m.	1.75
set1	6	114	947,951	1.36	5.25	1.16	7	147	15,040,942	40.21	o.o.m.	23.02
set2	6	140	2,762,420	3.55	28.70	3.16	7	189	78,130,591	452.43	o.o.m.	160.38
sor	14	66	3,188,645	9.16	32.29	0.22	16	76	28,697,813	174.48	o.o.m.	1.64
raytracer	9	121	4,882,833	10.36	42.57	0.54	10	132	24,414,083	98.15	o.o.m.	2.83
moldyn	13	83	3,188,633	8.66	23.77	0.22	15	93	28,697,831	166.83	o.o.m.	2.08
montecarlo	12	78	354,315	1.53	1.06	0.05	16	98	28,697,835	227.51	o.o.m.	1.88
hedc	7	92	458,334	0.64	1.50	0.38	9	121	24,522,560	108.37	o.o.m.	7.30
tsp	8	76	1,235,981	1.99	11.26	0.17	10	90	25,000,001	115.77	o.o.m.	52.33

If both H and F are not strongly feasible, then we get $G[1] \to G[2]$ and $G[2] \to G[1]$. Therefore, G contains the cycle $G[1] \to G[2] \to G[1]$, which is a contradiction to the assumption that G is strongly feasible. ◀

Moreover, in next section our experiments show that the gap between strong feasibility and reachability seldom exists in practice. We enumerate the reachable global states, by enumerating the strongly feasible global states, of losets that are captured from the execution of benchmark programs. In comparison with two naive but accurate enumeration algorithms, which simulate the execution of the program using one thread in a BFS or DFS fashion and hence only reachable global states are enumerated, our enumeration approach is able to produce exactly the same set of global states while using only 15–40% of their runtime.

6 Enumeration of Reachable Global State in the Loset Model

There are two approaches in literature to enumerate reachable global states of a computation. The first approach uses breadth (BFS) or depth (DFS) first strategy to add one event to the current global state G at a time [6, 12]. The event to be added satisfies the feasibility of G. This approach simulates the execution the program using one thread and hence every enumerated global state is reachable. Because DFS and BFS algorithms might enumerate the same global state more than once, this approach has to store the enumerated global states. In the worst case, the memory space for storing might grow exponentially in the number of threads in the computation.

An alternative approach predefines or calculates a spanning tree among the lattice of consistent global states and enumerates the global states following the edges of the tree [27, 18, 15, 11, 12, 4]. However, an edge may pass through unreachable global states because the set of consistent global states is a superset of reachable global states in a loset. Therefore, this approach needs to incorporate an additional function to prune the consistent but unreachable global states. In this paper, we use QuickLex [4] to enumerate the consistent global states and use strong feasibility to prune the unreachable global states.

Table 1 shows the information of the benchmarks that are used in the experiment. The benchmark *banking* is a toy program, which was used to demonstrates typical error patterns in concurrent programs [8]; *arraylist1* is a non-thread-safe container and *arraylist2* is a thread-safe container from Java library; *set1* and *set2* are implementations of concurrent sets using different fine-grained locking strategies [16]; *sor* is a scientific computation application; *raytracer*, *moldyn*, and *montecarlo* are parallel programs from Java Grande benchmark suite;

hedc is a crawler for searching Internet archives; and *tsp* is a parallel solver for the traveling salesman problem. The benchmarks *sor*, *raytracer*, *moldyn*, *montecarlo*, *hedc*, and *tsp* are the benchmark programs used in [5, 10, 32]. In addition, the columns of "n", "#events", and "#GS" show the number of threads, the number of events, and the number of enumerated global states of the computation, respectively.

All the experiments are conducted on a Linux machine with an Intel Xeon 2.67 GHz CPU and the heap size of Java virtual machine is limited to 2GB. The runtime is measured in seconds. Table 1 contains two sets of results. The set at the left of the table shows the largest computations that the DFS algorithm can handle, i.e., the DFS algorithm would run out of memory when the computations contain one more thread. On the other hand, the set at the right of the table shows the largest computations that the BFS algorithm can handle. The BFS and DFS algorithms generate the reachable global states and our approach generates strongly feasible global states. However, all the compared algorithms generate the same set of global states. Meanwhile, our approach reduces 84% and 61% of runtime in comparison with BFS and DFS algorithms, respectively.

7 Conclusion

In this paper, we present Loset, a model for a computation that contains locking constraints. We first show that the reachability problem in a loset is NP-complete. Afterwards, we present several useful properties of the model. Specifically, if a loset $\mathcal{L}$ is valid, then all lock-free feasible global states are reachable. In addition, the set of reachable lock-free feasible global states forms a distributive lattice. We also show that the reachability of G can be determined using only the subset of events that is located between G and the greatest lock-free feasible global state F that precedes G. Therefore, the set of lock-free feasible global state acts as a lower approximation and "reset" point of reachability.

We also present the property of strong feasibility, which is an upper approximation of reachability, and can be checked in polynomial time. The calculation is based on the inferred causality due to locking constraints and hence a reachable global state must be strongly feasible. Because of the lower and upper approximation of reachability, it is easy to answer the reachability of any given global state G in $\mathcal{L}$ if either G is lock-free feasible or not strongly feasible. If neither of these cases holds, then the reachability can be determined in the subcomputation $(G \backslash F)$ rather than the entire computation. Since our technique does not depend on the nature of predicates, it can be used for detecting the predicates whose nature are unknown and require the global view of the system.

References

1. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
2. Yen-Jung Chang. Predicate detection for parallel computations. In *PhD Dissertation, Department of Electrical and Computer Engineering, The University of Texas at Austin*, 2016.
3. Yen-Jung Chang and Vijay K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 140–149, 2015.
4. Yen-Jung Chang and Vijay K. Garg. Quicklex: A fast algorithm for consistent global states enumeration of distributed computations. In *International Conference On Principles of Distributed Systems*, 2015.

5 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.

6 R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

7 B. A. Davey and H. A. Priestley. Introduction to lattices and order. In *Cambridge University Press*, Cambridge, UK, 1990.

8 E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

9 Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*, pages 56–66, 1988.

10 Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of ACM SIGPLAN the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

11 Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

12 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.

13 Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.

14 Vijay K. Garg and B. Waldecker. Detection of unstable predicates. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, 1991.

15 Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.

16 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

17 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.

18 Roland Jegou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In *Proceedings of the International Workshop on Structures in Concurrency Theory*, pages 175–189, 1995.

19 Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of International Conference on Computer Aided Verification*, pages 505–518, 2005.

20 Vineet Kahlon and Chao Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of International Conference on Computer Aided Verification*, pages 434–449, 2010.

21 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

22 Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.

23 Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 125–226, Chateau de Bonas, France, 1988.

24 N. Mittal and V. K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.

25 Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation*, pages 446–455, 2007.

26 Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

27 Gara Pruesse and Frank Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.

28 S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 27–37, 1997.

29 Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

30 Ashis Tarafdar. Software fault tolerance in distributed systems using controlled re-execution. In *PhD Dissertation, Department of Electrical and Computer Engineering, The University of Texas at Austin*, 2000.

31 Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

32 Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

33 Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. *Formal Methods*, 29:256–272, 2009.

34 Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342, 2010.

WNetKAT: A Weighted SDN Programming and Verification Language*

Kim G. Larsen[1], Stefan Schmid[2], and Bingtian Xue[3]

1 **Aalborg University, Aalborg, Denmark**
`kgl@cs.aau.dk`
2 **Aalborg University, Aalborg, Denmark**
`schmiste@cs.aau.dk`
3 **Aalborg University, Aalborg, Denmark**
`bingt@cs.aau.dk`

Abstract

Programmability and verifiability lie at the heart of the software-defined networking paradigm. While OpenFlow and its match-action concept provide primitive operations to manipulate hardware configurations, over the last years, several more expressive network programming languages have been developed. This paper presents *WNetKAT*, the first network programming language accounting for the fact that networks are inherently weighted, and communications subject to capacity constraints (e.g., in terms of bandwidth) and costs (e.g., latency or monetary costs). *WNetKAT* is based on a syntactic and semantic extension of the NetKAT algebra. We demonstrate several relevant applications for *WNetKAT*, including cost- and capacity-aware reachability, as well as quality-of-service and fairness aspects. These applications do not only apply to classic, splittable and unsplittable (s, t)-flows, but also generalize to more complex (and stateful) network functions and service chains. For example, *WNetKAT* allows to model flows which need to traverse certain waypoint functions, which can change the traffic rate. This paper also shows the relationship between the equivalence problem of WNetKAT and the equivalence problem of the weighted finite automata, which implies undecidability of the former. However, this paper also shows the decidability of whether an expression equals to 0, which is sufficient in many practical scenarios, and we initiate the discussion of decidable subsets of the whole language.

1998 ACM Subject Classification C.2 Computer-Communication Networks

Keywords and phrases Software-Defined Networking, Verification, Reachability, Stateful Processing, Service Chains, Weighted Automata, Decidability, NetKAT

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.18

1 Introduction

Managing and operating traditional computer networks is known to be a challenging, manual and error-prone process. Given the critical role computer networks play today, not only in the context of the wide-area Internet but also of enterprise and data center networks, this is worrisome. Software-Defined Networks (SDNs) in general and the OpenFlow standard in particular, promise to overcome these problems by enabling automation, formal reasoning and verification, as well as by defining open standards for vendors. Indeed, there is also a wide consensus that formal verifiability is one of the key advantages of SDN over past attempts

* This work was partially supported by the ERC Advanced Grant *LASSO*, the Danish VILLUM project *ReNet* and the Sino-Danish Basic Research Center project *IDEA4CPS*.

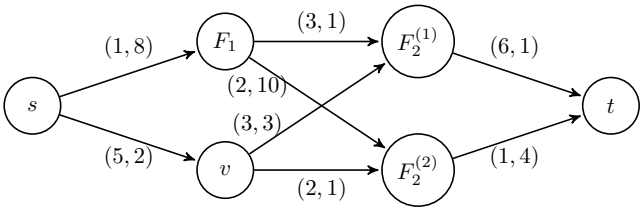


Figure 1 *Example:* A network hosting two (virtualized) functions F_1 and F_2. Function F_2 is allocated twice. The functions F_1 and F_2 may change the traffic rate.

to innovate computer networks, e.g., in the context of active networking [38]. Accordingly, SDN/OpenFlow is seen as a promising paradigm toward more dependable computer networks.

At the core of the software-defined networking paradigm lies the desire to program the network. In a nutshell, in an SDN, a general-purpose computer manages a set of programmable switches, by installing rules (e.g., for forwarding) and reacting to events (e.g., newly arriving flows or link failures). In particular, OpenFlow follows a match-action paradigm: the controller installs rules which define, using a *match* pattern (expressed over the *packet header fields*, and defining a *flow*), which packets (of a flow) are subject to which *actions* (e.g., forwarding to a certain port).

While the OpenFlow API is simple and allows to manipulate hardware configurations in flexible ways, it is very low level and not well-suited as a language for human programmers. Accordingly, over the last years, several more high-level and expressive domain-specific SDN languages have been developed, especially within the Frenetic project [13]. These languages can also be used to express fundamental network queries, for example related to *reachability*: They help administrators answer questions such as *"Can a given host A reach host B?"* or *"Is traffic between hosts A and B isolated from traffic between hosts C and D?"*.

What is missing today however is a domain-specific language which allows to describe the important *weighted aspects* of networking. E.g., real networks naturally come with capacity constraints, and especially in the Wide-Area Network (WAN) as well as in data centers, bandwidth is a precious resource. Similarly, networks come with latency and/or monetary costs: transmitting a packet over a wide-area link, or over a highly utilized link, may entail a non-trivial latency, and inter-ISP links may also be attributed with monetary costs.

Weights may not be limited to links only, but also nodes (switches or routers) have capacities and costs e.g., related to the packet rate. What is more, today's computer networks provide a wide spectrum of in-network functions related to security (e.g., firewalls) and performance (e.g., caches, WAN optimizers). To give an example, today, the number of so-called *middleboxes* in enterprise networks can be in the same order of magnitude as the number of routers [36]. A domain specific language for SDNs should be expressive enough to account for middleboxes which can change (e.g., compress or increase) the rate of the traffic passing through them. Moreover, a network language should be able to define that traffic must pass through these middleboxes in the first place, i.e., that routing policies fulfill waypointing invariants [39]. With the advent of more virtualized middleboxes, and the *Network Function Virtualization* paradigm, short *NFV*, (virtualized) middleboxes may also be *composed* to form more complex network services. For example, SDN traffic engineering flexibilities can be used to steer traffic through a series of middleboxes, concatenating the individual functions into so-called *service chains* [17, 26]. For instance, a network operator might want to ensure that all traffic from s to t should first be routed through a firewall FW, and then through a WAN optimizer WO, before eventually reaching t: the operator can do so by defining a service chain (s, FW, WO, t).

Let us consider a more detailed example, see the network in Figure 1: The network hosts two types of (virtualized) functions F_1 and F_2: possible network functions may include, e.g., a firewall, a NAT, a proxy, a tunnel endpoint, a WAN optimizer (and its counterpart), a header decompressor, etc. In this example, function F_2 is instantiated at two locations. Functions F_1 and F_2 may not be flow-preserving, but may *decrease* the traffic rate (e.g., in case of a proxy, WAN optimizer, etc.) or *increase* it: e.g., a tunnel entry-point may add an extra header, a security box may add a watermark to the packet, the counterpart of the WAN optimizer may decompress the packet, etc. Links come with a certain cost (say latency) and a certain capacity (in terms of bandwidth). Accordingly, we may annotate links with two weights: the tuple $(2, 3)$ denotes that the link cost is 2 and the link capacity 3. We would like to be able to ask questions such as: *Can source s emit traffic into the service chain at rate x without overloading the network?* or *Can we embed a service chain of cost (e.g., end-to-end latency) at most x?*.

1.1 Contributions

This paper initiates the study of weighted network languages for programming and reasoning about SDN networks, which go beyond topological aspects but account for actual resource availabilities, capacities, costs, or even stateful operations. In particular, we present *WNetKAT*, an extension of the *NetKAT* [1] algebra.

For example, *WNetKAT* supports a natural generalization of the reachibility concepts used in classic network programming languages, such as *cost-aware* or *capacity-aware reachability*. In particular, *WNetKAT* allows to answer questions of the form: *Can host A reach host B at cost/bandwidth/latency x?*

We demonstrate applications of *WNetKAT* for a number of practical use cases related to performance, quality-of-service, fairness, and costs. These applications are not only useful in the context of both splittable and unsplittable routing models, where flows need to travel from a source s to a destination t, but also in the context of more complex models with waypointing requirements (e.g., service chains).

The weighted extension of NetKAT is non-trivial, as capacity constraints introduce dependencies between flows, and arithmetic operations such as *addition* (e.g., in case of latency) or *minimum* (e.g., in case of bandwidth to compute the end-to-end delay) have to be supported along the paths. Therefore, we extend the syntax of NetKAT toward weighted packet- and switch-variables, as well as queues, and provide a semantics accordingly. In particular, one contribution of our work is to show for which weighted aspects and use cases which language extensions are required.

We also show the relation between WNetKAT expressions and weighted finite automata [9] – an important operational model for weighted programs. This leads to the undeciability of WNetKAT equivalence problem. However, leveraging this relation we also succeed to prove the decidability of whether an expression equals to 0: for many practical scenarios a sufficient and relevant solution. Moreover, this paper initiates the discussion of identifying decidable subsets of the whole language.

1.2 Related Work

Most modern domain-specific SDN languages enable automated tools for verifying network properties [12, 13, 29, 43, 44]. Especially reachability properties, which are also the focus in our paper, have been studied intensively in the literature [19, 20]. Indeed, the formal verifiability of the OpenFlow match-action interface [19, 20, 30, 46] constitutes a key advantage

of the paradigm over previous innovation efforts [5]. Existing expressive languages use SAT formulas [27], graph-based representations [19, 20], or higher-order logic [45] to describe network topologies and policies.

Our work builds upon NetKAT, a new framework based on Kleene algebra with tests for specifying, programming, and reasoning about networks and policies. NetKAT respresents a more principled approach compared to prior work, and is also motivated by the observation that end-to-end functionality is determined not only by the behavior of the switches and but also by the structure of the network topology. NetKAT in turn is based on earlier efforts performed in the context of NetCore [28], Pyretic [29] and Frenetic [13]. It has recently been extended to a probabilistic [14] and temporal [2] setting, and first versions for specific use cases like QoS are emerging [34]. The Kleene algebra with tests was developed by Kozen [24]. However, to the best of our knowledge, there is no prior work on a general weighted (and stateful) version of NetKAT.

In general, stateful network design and analysis is very active field of research, and there are several interesting recent results, e.g., on the complexity and scalability of more stateful verification [42], on quantitative analysis [18], or on reachability [10].

2 Background

A Software-Defined Network (SDN) outsources and consolidates the control over data plane elements to a logically centralized control plane implemented in software. Arguably, software-defined networking in general, and its de facto standard, OpenFlow, are about programmability, verifiability and generality [11]: The behavior of an OpenFlow switch is defined by its configuration: a list of prioritized *(flow) rules* stored in the switch flow table, which are used to classify, filter, and modify packets based on their *header fields*. In particular, OpenFlow follows a simple match-action paradigm: the match parts of the flow rules (expressed over the header fields) specify which packets belong to a certain flow (e.g., depending on the IP destination address), and the action parts define how these packets should be processed (e.g., forward to a certain port). OpenFlow supports a rather general packet processing: it allows to match and process packets based on their Layer-2 (e.g., MAC addresses), Layer-3 (e.g., IP addresses), and Layer-4 header fields (e.g., TCP ports), or even in a protocol-independent manner, using arbitrary bitmasking [4].

OpenFlow also readily supports quantitative aspects, e.g., the selection of queues annotated with different round robin weights (the standard approach to implement quality-of-service guarantees in networks today), or meters (measuring the bandwidth of a flow). Moreover, we currently witness a trend toward more flexible and stateful programmable switches and packet processors, featuring group tables, counters, and beyond [3].

The formal framework developed in this paper is based on NetKAT [1]. NetKAT is a high-level algebra for reasoning about network programs. It is based on *Kleene Algebra with Tests (KAT)*, and uses an equational theory combining the axioms of KAT and network-specific axioms that describe transformations on packets (as performed by OpenFlow switch rules). These axioms facilitate reasoning about local switch processing functionality (needed during compilation and for optimization) as well as global network behavior (needed to check reachability and traffic isolation properties). Basically, an atomic NetKAT policy (a function from packet headers to sets of packet headers: essentially the per-switch OpenFlow rules discussed above) can be used to filter or modify packets. Policy combinators ($+$) allow to build larger policies out of smaller policies. There is also a sequential composition combinator to apply functions consecutively.

Besides the *policy*, modeling the per-switch OpenFlow rules, a network programming language needs to be able to describe the network *topology*. NetKAT models the network topology as a directed graph: nodes (hosts, routers, switches) are connected via edges (links) using (switch) *ports*. NetKAT simply describes the topology as the union of smaller policies that encode the behavior of each link. To model the effect of sending a packet across a link, NetKAT employs the sequential composition of a filter that retains packets located at one end of the link, and a modification that updates the switch and port fields to the location at the other end of the link. Note that the NetKAT topology and the NetKAT policy are hence to be seen as two independent concepts. Succinctly:

A *Kleene algebra* (KA) is any structure $(K, +, \cdot, ^*, 0, 1)$, where K is a set, $+$ and $\cdot$ are binary operations on K, * is a unary operation on K, and 0 and 1 are constants, satisfying the following axioms, where we define $p \leq q$ iff $p + q = q$.

$$
\begin{array}{ll}
p + (q + r) = (p + q) + r & p(qr) = (pq)r \\
p + q = q + p & 1 \cdot p = p \cdot 1 = p \\
p + 0 = p + p = p & p \cdot 0 = 0 \cdot p = 0 \\
p(q + r) = pq + pr & (p + q)r = pr + qr \\
1 + pp^* \leq p^* & q + px \leq x \Rightarrow p^*q \leq x \\
1 + p^*p \leq p^* & q + xp \leq x \Rightarrow qp^* \leq x
\end{array}
$$

A *Kleene algebra with tests* (KAT) is a two-sorted structure $(K, B, +, \cdot, ^*, \text{-}, 0, 1)$, where $B \subseteq K$ and

- $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra;
- $(B, +, \cdot, \text{-}, 0, 1)$ is a Boolean algebra;
- $(B, +, \cdot, 0, 1)$ is a subalgebra of $(K, +, \cdot, 0, 1)$.

The elements of B are called *tests*. The axioms of Boolean algebra are:

$$
\begin{array}{ll}
a + bc = (a + b)(a + c) & ab = ba \\
a + 1 = 1 & a + \overline{a} = 1 \\
a\overline{a} = 0 & aa = a
\end{array}
$$

NetKAT is a version of KAT in which the atoms (elements in K) are defined over header fields f (variables) and values ω:

- $f \leftarrow \omega$ ("assign a value ω to header field f")
- $f = \omega$ ("test the value of a header field")
- dup ("duplicate the packet")

The set of all possible values of f is denoted Ω. For readability, we use *skip* and *drop* to denote 1 and 0, respectively.

The NetKAT axioms consist of the following equations, in addition to the KAT axioms on the commutativity and redundancy of different actions and tests, and enforcing that the field has exactly one value:

$$
\begin{array}{rcll}
f_1 \leftarrow \omega_1; f_2 \leftarrow \omega_2 & = & f_2 \leftarrow \omega_2; f_1 \leftarrow \omega_1 \quad (f_1 \neq f_2) & (1) \\
f_1 \leftarrow \omega_1; f_2 = \omega_2 & = & f_2 = \omega_2; f_1 \leftarrow \omega_1 \quad (f_1 \neq f_2) & (2) \\
f = \omega; \mathsf{dup} & = & \mathsf{dup}; f = \omega & (3) \\
f \leftarrow \omega; f = \omega & = & f \leftarrow \omega & (4) \\
f = \omega; f \leftarrow \omega & = & f = \omega & (5) \\
f \leftarrow \omega_1; f \leftarrow \omega_2 & = & f \leftarrow \omega_2 & (6) \\
f = \omega_1; f = \omega_2 & = & 0 \quad (\omega_1 \neq \omega_2) & (7) \\
\sum_{\omega \in \Omega} f = \omega & = & 1 & (8)
\end{array}
$$

In terms of semantics, NetKAT uses *packet histories* to record the state of each packet on its path from switch to switch through the network. The notation $\langle pk_1, \ldots, pk_n \rangle$ is used to describe a history with elements $pk_1, \ldots, pk_n$ being packets; $pk :: \langle\rangle$ is used to denote a history with one element and $pk :: h$ to denote the history constructed by prepending pk on to h. By convention, the first element of a history is the current packet (the "head"). A NetKAT expression denotes a function $[\![\ ]\!] : H \to 2^H$, where H is the set of packet histories. Histories are only needed for reasoning: Policies only inspect or modify the first (current) packet in the history. Succinctly:

$$
\begin{aligned}
[\![f \leftarrow \omega]\!](pk :: h) &= \{pk[\omega/f] :: h\} \\
[\![f = \omega]\!](pk :: h) &= \begin{cases} \{pk :: h\} & \text{if } pk(f) = \omega \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\mathsf{dup}]\!](pk :: h) &= \{pk :: pk :: h\} \\
[\![p + q]\!](h) &= [\![p]\!](h) \cup [\![q]\!](h) \\
[\![pq]\!](h) &= \bigcup_{h' \in [\![p]\!](h)} [\![q]\!](h') \\
[\![p^*]\!](h) &= \bigcup_n [\![p^n]\!](h) \\
[\![0]\!](h) &= \emptyset \\
[\![1]\!](h) &= \{h\} \\
[\![\overline{a}]\!](h) &= \begin{cases} \{h\} & \text{if } [\![a]\!](h) = \emptyset \\ \emptyset & \text{if } [\![a]\!](h) = \{h\} \end{cases}
\end{aligned}
$$

▶ **Example 1.** Consider the network in Figure 1. NetKAT can be used to specify the topology as follows, where the field sw stores the current location (switch) of the packet:

$$
\begin{aligned}
t \quad ::= \quad & sw = s; (sw \leftarrow F_1 + sw \leftarrow v) \\
& + sw = F_1; (sw \leftarrow F_2^{(1)} + sw \leftarrow F_2^{(2)}) \\
& + sw = v; (sw \leftarrow F_1^{(1)} + sw \leftarrow F_2^{(2)}) \\
& + sw = F_2^{(1)}; sw \leftarrow t \\
& + sw = F_2^{(2)}; sw \leftarrow t
\end{aligned}
$$

The first line of the above NetKAT expression specifies that if the packet is at s, then it will be sent to F_1 or v. Analogously for the other cases. In OpenFlow, this policy can be implemented using OpenFlow rules, whose match part applies to packets arriving at s, and whose action part assigns the packets to the respective forwarding ports.

However, one can observe that with NetKAT it is not possible to specify or reason about the important quantitative aspects in Figure 1, e.g., the cost and capacity along the links or the function of F_2 which changes the rate of the flow. To do these, a weighted extension of NetKAT is needed.

3 WNetKAT

On a high level, a computer network can be described as a set of nodes (hosts or routers) which are interconnected by a set of links, hence defining the network topology. While this high-level view is sufficient for many purposes, for example for reasoning about reachability, in practice, the situation is often more complex: both nodes and links come with capacity constraints (e.g., in terms of buffers, CPU, and bandwidth) and may be attributed with costs (e.g., monetary or in terms of performance). In order to reason about performance, cost, and fairness aspects, it is therefore important to take these dimensions into account.

The challenge of extending NetKAT to weighted scenarios lies in the fact that in a weighted network, traffic flows can no longer be considered independently, but they may *interfere*:

their packets compete for the shared resource. Moreover, packets of a given flow may not necessarily be propagated along a unique path, but may be split and distributed among multiple paths (in the so-called *multi-path routing* or *splittable flow* variant). Accordingly, a weighted extension of NetKAT must be able to deal with "inter-packet states".

We in this paper will think of the network as a weighted (directed) graph $G = (V, E, w)$. Here, V denotes the set of switches (or equivalently routers, and henceforth often simply called nodes), E is the set of links (connected to the switches by *ports*), and w is a weight function. The weight function w applies to both nodes V as well as links E. Moreover, a node and a link may be characterized by *a vector of weights* and also combine *multiple resources*: for example, a list of capacities (e.g., CPU and memory on nodes, or bandwidth on links) and a list of costs (e.g., performance, energy, or monetary costs).

In order to specify the quantitative aspects, we propose in this paper a weighted extension of NetKAT: *WNetKAT*. In addition to NetKAT:

- *WNetKAT* includes a set of *quantitative packet-variables* to specify the quantitative information carried in the packet, in addition to the regular (non-quantitative) packet-variables of NetKAT (called *fields* in NetKAT): e.g., regular variables are used to describe locations, such as switch and port, or priorities, while quantitative variables are used to specify latency or energy. The set of all packet-variables is denoted by $\mathcal{V}_p$.
- *WNetKAT* also includes a set of *switch-variables*, denoted by $\mathcal{V}_s$, to specify the configurations at the (stateful) switch. Switch variables can either be quantitative (e.g., counters, meters, meta-rules [4, 33]) or non-quantitative (e.g., location related), as it is the case of the packet-variables.

▶ Remark. The set of quantitative (packet- and switch-) variables is denoted by $\mathcal{V}_q$ and these variables range over the natural numbers $\mathbb{N}$ (e.g., normalized rational numbers). The set of non-quantitative (packet- and switch-) variables is denoted $\mathcal{V}_n$ and the set of the possible values is denoted Ω. Note that $\mathcal{V}_q \cap \mathcal{V}_n = \emptyset$ and $\mathcal{V}_q \cup \mathcal{V}_n = \mathcal{V}_p \cup \mathcal{V}_s$.

In addition to introducing quantitative variables, we also need to extend the atomic actions and tests of NetKAT. Concretely, *WNetKAT* first supports non-quantitative assignments and non-quantitative tests on the non-quantitative switch-variables, similar to those on the packet-variables in NetKAT. Moreover, *WNetKAT* also allows for *quantitative assignments* and *quantitative tests*, defined as follows, where $x \in \mathcal{V}_q$, $\mathcal{V}' \subseteq \mathcal{V}_q$, $\delta \in \mathbb{N}$, $\bowtie \in \{>, <, \leq, \geq, =\}$:

- **Quantitative Assignment.** $x \leftarrow (\Sigma_{x' \in \mathcal{V}'} x' + \delta)$: Read the current values of the variables in $\mathcal{V}'$ and add them to δ, then assign this result to x.
- **Quantitative Test.** $x \bowtie (\Sigma_{x' \in \mathcal{V}'} x' + \delta)$: Read the current value of the variables in $\mathcal{V}'$ and add them to δ, then compare this result to the current value of x.

▶ Remark.

1. In the quantitative assignment and test, only addition is allowed. However, an extension to other arithmetic operations (e.g., linear combinations) is straightfoward. Moreover, calculating *minimum* or *maximum* may be useful in practice: e.g., the throughput of a flow often depends on the weakest link (of minimal bandwidth) along a path. Note that these operations can actually be implemented with quantitative assignments and tests, i.e., by comparing every variable to another and determining the smallest. E.g., for $x \in \mathcal{V}_q$ and $y, z \in \mathcal{V}_q$ or $\mathbb{N}$,

$$x \leftarrow \min\{y, z\} \stackrel{\text{def}}{=} y \leq z; x \leftarrow y \;\&\; y > z; x \leftarrow z\,.$$

2. In quantitative assignment and test, x might be in $\mathcal{V}'$.
3. We use $+$ to denote the arithmetic operation over numbers. Therefore, we will use "&" in *WNetKAT* to denote the "+" operator of Kleene Algebra, which is also used in [14].

Table 1 Semantics of *WNetKAT*.

$$[\![x \leftarrow \omega]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk[\omega/x] :: h\} & \text{if } x \in \mathcal{V}_p \\ \{\rho(v)[\omega/x],\ pk :: h\} & \text{if } x \in \mathcal{V}_s \text{ and } pk(sw) = v \end{cases} \quad (1)$$

$$[\![x = \omega]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk :: h\} & \text{if } x \in \mathcal{V}_p \text{ and } pk(x) = \omega \\ & \text{or if } x \in \mathcal{V}_s, pk(sw) = v \text{ and } \rho(v,x) = \omega \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

$$[\![y \leftarrow (\Sigma_{y' \in \mathcal{V}'} y' + r)]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk[r'/x] :: h\} & \text{if } x \in \mathcal{V}_p \\ \{\rho(v)[r'/x],\ pk :: h\} & \text{if } x \in \mathcal{V}_s \text{ and } pk(sw) = v \end{cases} \quad (3)$$
where $r' = \Sigma_{y_p \in \mathcal{V}' \cap \mathcal{V}_p} pk(y_p) + \Sigma_{y_s \in \mathcal{V}' \cap \mathcal{V}_q} \rho(v, y_s) + r$

$$[\![y = (\Sigma_{y' \in \mathcal{V}'} y' + r)]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk :: h\} & \text{if } x \in \mathcal{V}_p \text{ and } pk(x) = r' \\ & \text{or } x \in \mathcal{V}_s, pk(sw) = v \text{ and } \rho(v,x) = r' \\ \emptyset & \text{otherwise} \end{cases} \quad (4)$$
where $r' = \Sigma_{y_p \in \mathcal{V}' \cap \mathcal{V}_p} pk(y_p) + \Sigma_{y_s \in \mathcal{V}' \cap \mathcal{V}_q} \rho(v, y_s) + r$

Given the set of switches V, a *switch-variable valuation* is a partial function $\rho : V \times \mathcal{V}_s \hookrightarrow \mathbb{N} \cup \Omega$. It associates, for each switch and each switch-variable, a integer or a value from Ω. We emphasize that ρ is a partial function, as some variables may not be defined at some switches.

A *WNetKAT*expression denotes a function $[\![\ ]\!] : \rho \times H \to 2^H$, where H is the set of packet histories. The semantics of *WNetKAT* is defined in Table 1, where $x \in \mathcal{V}_n, y \in \mathcal{V}_q$, $\delta \in \mathbb{N}$ and $\omega \in \Omega$.

▶ Remark.

- Equations (1) and (3) update the corresponding header field if x is a packet-variable, or they update the corresponding switch information of the current switch if x is a switch-variable. Equation (1) updates the non-quantitative variables and Equation (3) the quantitative ones.
- Equations (2) and (4) test the non-quantitative and quantitative variables respectively, using the current packet- and switch-variables.

▶ **Example 2.** Consider again the network in Figure 1. The topology of the network can be characterized with the following *WNetKAT* formula t, where sw specifies the current location (switch) of the packet, co specifies the cost, and ca specifies the capacity along the links.

$$\begin{aligned} t ::= \quad & sw = s; (sw \leftarrow F_1; co \leftarrow co + 1; ca \leftarrow \min\{ca, 8\} \\ & \qquad \& \ sw \leftarrow v; co \leftarrow co + 5; ca \leftarrow \min\{ca, 2\}) \\ & \& \ sw = F_1; \\ & \qquad (sw \leftarrow F_2^{(1)}; co \leftarrow co + 3; ca \leftarrow \min\{ca, 1\} \\ & \qquad \& \ sw \leftarrow F_2^{(2)}; co \leftarrow co + 2; ca \leftarrow \min\{ca, 10\}) \\ & \& \ sw = v; (sw \leftarrow F_2^{(1)}; co \leftarrow co + 3; ca \leftarrow \min\{ca, 3\} \\ & \qquad \& \ sw \leftarrow F_2^{(2)}; co \leftarrow co + 2; ca \leftarrow \min\{ca, 1\}) \\ & \& \ sw = F_2^{(1)}; sw \leftarrow t; co \leftarrow co + 6; ca \leftarrow \min\{ca, 1\} \\ & \& \ sw = F_2^{(2)}; sw \leftarrow t; co \leftarrow co + 1; ca \leftarrow \min\{ca, 4\} \end{aligned}$$

The variable co accumulates the costs along the path, and the variable ca records the smallest capacity along the path. Notice that ca is just a packet-variable used to record the capacity of the path; it does not represent the capacity used by this packet (the latter is assumed to be negligible).

Assume that function F_1 is flow conserving (e.g., a NAT), while F_2 increases the flow rate by an additive constant $\gamma \in \mathbb{N}$ (e.g., a security related function, adding a watermark or

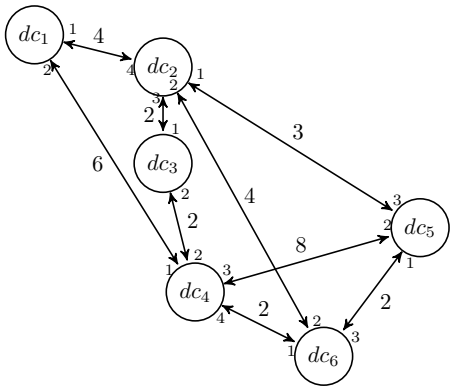


Figure 2 Example topology: excerpt of Google B4 [16] (U.S. data centers only). Nodes here represent data centers (resp. OpenFlow switches located at the end of the corresponding long-haul fibers). Links are annotated with weights, and nodes are interconnected via ports (*small numbers*).

an IPSec header). The policy of F_2 can be specified as:

$$p_{F_2} ::= \quad (sw = F_2^{(1)} \ \& \ sw = F_2^{(2)}); ca \leftarrow ca + \gamma$$

▶ Remark. Note that this simple example required only (non-quantitative and quantitative) packet-variables. However, as we will see in Section 4, to model more complex aspects of networking, such as splittable flows, additonal concepts of *WNetKAT* will be needed.

4 Applications

The weighted extensions introduced by *WNetKAT* come with a number of interesting applications. In this section, we show that the notions of reachability frequently discussed in prior work, find natural extensions in the world of weighted networks, and discuss applications in the context of service chains, fairness, and quality-of-service. In our technical report [25], additional details are provided for some of these use cases.

4.1 Cost Reachability

Especially data center networks but also wide-area networks, and to some extent enterprise networks, feature a certain *path diversity* [41]: there exist multiple routes between two endpoints (e.g., hosts). This path diversity is not only a prerequisite for fault-tolerance, but also introduces traffic engineering flexibilities. In particular, different paths or routes depend on different links, whose cost can vary. For example, links may be attributed with monetary costs: a peering link may be free of charge, while an up- or down-link is not. Links cost can also be performance related, and may for example vary in terms of latency, for example due to the use of different technologies [37], or simply because of different physical distances. The monetary and performance costs are often related: for example, in the context of stock markets, lower latency links come at a higher price [35]. It is therefore natural to ask questions such as: *"Can A reach B at cost at most c?"*. We will refer to this type of questions as *cost reachability questions*.

▶ **Example 3.** Consider the network in Figure 2. The topology roughly describes the North American data centers interconnected by Google B4, according to [16].

In order to reason about network latencies, we not only need information about the switch at which the packet is currently located (as in our earlier examples), but also the *port of the switch* needs to be specified. We introduce the packet-variable pt. We can then specify this network topology in *WNetKAT*. The link from dc_1 to dc_2 (latency 4 units) represented by the port 1 at dc_1 and the port 4 at dc_2 is specified as follows, where we use packet-variable sw to denote the current switch, pt to specify the current port, and l to specify the latency of the path the packet traverses,

$$sw = dc_1; pt = 1; sw \leftarrow dc_2; pt \leftarrow 4; l \leftarrow l + 4\,.$$

Analogously, the entire network topology can be modeled with *WNetKAT*, henceforth denoted by t. The policy of the network determines the functionality of each switch (the OpenFlow rules), e.g., in dc_2, packets from dc_1 to dc_5 arriving at port 4 are always sent out through port 1 or port 3. This can be specified as:

$$src = dc_1; dst = dc_5; sw = dc_2; pt = 4; (pt \leftarrow 1 \,\&\, pt \leftarrow 3)\,.$$

Analogously, the entire network policy can be modeled with *WNetKAT*, henceforth denoted by p.

To answer the cost reachability question, one can check whether the following *WNetKAT* expression is equal to *drop*.

$$scr \leftarrow A; dst \leftarrow B; l \leftarrow 0; sw \leftarrow X; pt(pt)^*; sw = B; l \leq c\,.$$

If it is equal to *drop*, then B cannot be reached from A at latency at most c; otherwise, it can.

▶ Remark. For ease of presentation, in the above example, we considered only one weight. However, *WNetKAT* readily supports multiple weights: we can simply use multiple variables accordingly. Moreover, while the computational problem complexity can increase with the number of considered weights [23], the multi-constrained path selection does not affect the general asymptotic complexity of *WNetKAT*.

4.2 Capacitated Reachability

Especially in the wide-area network, but also in data centers, link capacities are a scarce resource: indeed, wide-area traffic is one of the fastest growing traffic aggregates [16]. However, also the routers themselves come with capacity constraints, both in terms of memory (size of TCAM) as well as CPU: for example, the CPU utilization has been shown to depend on the packet rate [31]. Accordingly, a natural question to ask is: *Can A communicate at rate at least r to B?* We will refer to this type of questions as *capacitated reachability questions*.

There are two problem variants:

- *Unsplittable flows:* The capacity needs to be computed along a single path (e.g., an MPLS tunnel).
- *Splittable flows:* The capacity needs to be computed along multiple paths (e.g., MPTCP, ECMP). We will assume links of higher capacity are chosen first.

For both variants, to find out the capacity of paths between two nodes, a *single* test packet will be sent to explore the network and record the bandwidth/capacity with a packet-variable in the packet. We assume that the bandwidth consumed by this packet is negligibile. Also, only once the packet has traversed and determined the bandwidth, e.g., the actual (large) flows are allocated accordingly (by the SDN controller).

▶ **Example 4.** Consider the network in Figure 2 again, but assume that the labels are the capacities rather than latency.

Unsplittable flow scenario: The switch policies are exactly the same as in Example 3, while the topology will be specified similarly using packet-variable c to record the capacity of the link. E.g., the link between dc_1 and dc_2 can be specified as:

$$sw = dc_1; pt = 1; sw \leftarrow dc_2; pt \leftarrow 4; c \leftarrow \min\{c, 4\}\,.$$

The unsplittable capacitated reachability question can be answered by checking whether the following expression is equal to *drop*,

$$scr \leftarrow A; dst \leftarrow B; c \leftarrow r; sw \leftarrow A; pt(pt)^*; sw = B; c \geq r\,.$$

If the above formula does not equal *drop*, then A can communicate at rate at least r to B.

Another (possibly) more efficient approach is not to update c while the bandwidth is smaller than r (meaning that a flow of size r cannot go through this link). In this case, one can specify the topology as follows, where c is not used to record the capacity along the path anymore, but rather to test whether this link is wide enough:

$$sw = dc_1; pt = 1; sw \leftarrow dc_2; pt \leftarrow 4; c \leq 4\,.$$

The above *WNetKAT* expression only tests whether c is less than or equal to 4. It makes sure that the value of c (which is r) does not exceed the capacity of the following link. If it exceeds the capacity of the link, then a flow of rate r cannot use this link. Therefore, the test packet is dropped already. The capacitated reachability question can then be answered by checking whether the following expression is equal to *drop*:

$$scr \leftarrow A; dst \leftarrow B; c \leftarrow r; sw \leftarrow A; pt(pt)^*; sw = B\,.$$

If the above formula does not equal *drop*, then A can communicate at rate at least r to B.

Splittable flow scenario: For the splittable scenario, the situation is far more complicated. For example, in dc_2, packets arriving at port 4 are sent out through port 1 or port 2, and port 2 prioritizes port 1. That is, if the incoming traffic has rate 4, then a share of 3 units will be sent out through port 2, and a 1 share through port 1.

Note that also here, still only one *single* test packet will be sent to collect the capacity information. This information will be stored in the packet-variable c as well. However, when the test packet arrives at a switch where a flow can be split, copies of the packet are sent (after updating the c according to the bandwidth of each path) to all possible paths, to record the capacity along all other paths. This exploits the fact that *WNetKAT* (NetKAT) treats the & operator as *conjunction* in the sense that both operations are performed, rather than *disjunction*, where one of the two operations would be chosen non-deterministically (according to the usual Kleene interpretation). Again, we emphasize that we will refer to c stored in one *single* test packet, and not the actual real data flow. Now the topology will update c as in the unsplittable case. However, the policy needs to not only decide which ports the packets go to, but also update c according to the split policy. E.g.,, at dc_2, the data flow from dc_1 to dc_5 at rate 4 is sent out through port 1 at rate 3, and the port 3 at 1. And if the rate is smaller than or equal to 3, e.g., 2, then the whole flow of rate 2 will be sent out through port 1. The following *WNetKAT* formula specifies this behavior:

$$\begin{aligned} src = dc_1; dst = dc_5; sw = dc_2; pt = 4; c \leq 5 \\ (pt \leftarrow 1; c \leftarrow \min\{3, c\} \\ \&\ pt \leftarrow 3; c \leftarrow \max\{0, c-3\}) \end{aligned}$$

The test $c \leq 5$ ensures that the flow does not exceed the capacity of both paths. Notice that even when the size of the flow is small enough for one path, a copy of the test packet with $c = 0$ will still be sent to the other. This ensures that sufficient information is available at the switch where flows merge. That is, the switch collects the weights the packets carry (c in our example). The switch will only push packets to the right out-ports after all expected packets have arrived. This will happen before the switch sends the packet to the right out-ports. For example, at dc_4, the flow from dc_1 to dc_5 might arrive in from ports 1 and 2 and will be sent out through port 3. In order to record the capacity of both links, switch-variables C and X are introduced, for each possible merge. For example, the following table provides the merging rules for the switch at dc_4, where X is the counter for the merge, and C stores the current capacity of the arriving test packets. Initially, X is set to the number of in-ports for the merge, and C is set to 0.

src	dst	in	out	C	X
dc_1	dc_5	$1, 2$	3	0	2
dc_5	dc_2	$3, 4$	$1, 2$	0	2

The first line of the rules in the table can be specified in *WNetKAT* as follows:

$$\begin{aligned} &sw = dc_4; src = dc_1; dst = dc_5; (pt = 1 \;\&\; pt = 2); \\ &\qquad C \leftarrow C + c; X \leftarrow X - 1; \\ &\quad (X \neq 0; drop \;\&\; X = 0; c \leftarrow C; pt \leftarrow 3) \end{aligned}$$

When a packet from dc_1 to dc_5 arrives at port 1 or 2 of dc_4, first the switch collects the value of c and adds it to the switch-variable C, then decrements X to record that one packet arrived. Afterwards, we test whether all expected packets arrived ($X = 0$). If not, the current one is dropped; if yes, we send the current packet out to port 3. The reason that we can drop all packets except for the last, is that all those packets carry exactly the same values. Therefore, we eventually only need to include the merged capacity (C) in the last packet, and propagate it.

Combining the split and merge cases, the policy of the switch can be defined. For example, the second line of the merging rule table can be specified as follows, by first merging from port 3 and 4, and then splitting to port 1 and 2:

$$\begin{aligned} &sw = dc_4; src = dc_5; dst = dc_2; (pt = 3 \;\&\; pt = 4); \\ &\quad C \leftarrow C + c; X \leftarrow X - 1; \\ &\quad (X \neq 0; drop \;\&\; X = 0; c \leftarrow C; c \leq 8 \\ &\qquad\qquad (pt \leftarrow 1; c \leftarrow \min\{6, c\} \\ &\qquad\qquad \&\; pt \leftarrow 2; c \leftarrow \max\{0, c - 6\})) \end{aligned}$$

Then the splittable capacited reachability question can be answered by checking whether the following expression evaluates to *drop*:

$$\begin{aligned} &scr \leftarrow A; dst \leftarrow B; c \leftarrow r; sw \leftarrow A; pt(pt)^*; \\ &\qquad sw = B; X = 0; c \geq r \end{aligned}$$

If the above formula does not equal *drop*, then A can communicate at rate at least r to B.

4.3 Service Chaining

The virtualization and programmability trend is not limited to the network, but is currently also discussed intensively for network functions in the context of the Network Function

Virtualization (NFV) paradigm. SDN and NFV nicely complement each other, enabling innovative new network services such as *service chains* [17]: network functions which are traversed in a particular order (e.g., first firewall, then cache, then wide-area network optimizer). Our language allows to reason about questions such as *Are sequences of network functions traversed in a particular order, without violating node and link capacities? WNetKAT* can easily be used to describe weighted aspects also in the context of service chains. In particular, network functions may both *increase* (e.g., due to addition of an encapsulation header, or a watermark) or *decrease* (e.g., a WAN optimizer, or a cache) the traffic rate, both *additively* (e.g., adding a header) or *multiplicatively* (e.g., WAN optimizer).

▶ **Example 5.** Let us go back to Figure 1, and consider a service chain of the form (s, F_1, F_2, t): traffic from s to t should first traverse a function F_1 and then a function F_2, before reaching t. For example, F_1 may be a firewall or proxy and F_2 is a WAN optimizer. The virtualized functions F_1 and F_2 may be allocated redundantly and may change the traffic volume. Using *WNetKAT*, we can ask questions such as: *What is the maximal rate at which s can transmit traffic into the service chain?* or *Can we realize a service chain of cost (e.g., latency) at most x?*. Let us consider the following example: The question *"Can s reach t at cost/latency at most ℓ and/or at rate/bandwidth at least r, via the service chain functions F_1 and F_2?"*, can be formulated by combining the reachability problems above and the waypointing technique in [1]. For example, in case of cost reachability, we can ask if the following *WNetKAT* formula equals *drop*.

$$\begin{aligned} &src \leftarrow s; dst \leftarrow t; co \leftarrow 0; sw \leftarrow s; pt(pt)^*; \\ &\quad sw = F_1; p_{F_1}; tpt(pt)^*; sw = F_2; p_{F_2}; \\ &\qquad tpt(pt)^*; sw = t; co \leq \ell; ca \geq r \end{aligned}$$

Note that in this example, we considered an unsplittable scenario. For the splittable scenario, we can extend the splittable capacitated reachability use case above analogously.

5 (Un)Decidability

In this section we shed light on the fundamental decidability of weighted SDN programming languages like WNetKAT. Given today's trend toward more quantitative networking, we believe that this is an important yet hardly explored dimension. In particular, we will establish an equivalence between WNetKAT and weighted automata.

In the following, we will restrict ourselves to settings where quantitative variables of the same type behave similarly in the entire network: For example, the cost variables (e.g., quantifying latencies) in the network are always added up along a given path, while capacity variables require minimum operations along different paths. This is a reasonable for real-world networks.

The definition of the weighted automata used here is slightly different from those usually studied, e.g., [6, 9]. However, it is easy to see that they are equivalent.

We first introduce some preliminaries. A *semiring* is a structure $(K, \oplus, \otimes, 0, 1)$, where $(K, \oplus, 0)$ is a commutative monoid, $(K, \otimes, 1)$ is a monoid, multiplication distributes over addition $k \otimes (k' \oplus k'') = k \otimes k' \oplus k \otimes k''$, and $0 \otimes k = k \otimes 0 = 0$ for each $k \in K$. For example, $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ and $(\mathbb{N} \cup \{\infty\}, \max, +, \infty, 0)$ are semirings, named the *tropical semiring*. $(\mathbb{N} \cup \{\infty\}, \max, \min, 0, \infty)$ is also a semiring. A *bimonoid* is a structure $(K, \oplus, \otimes, 0, 1)$, where $(K, \oplus, 0)$ and $(K, \otimes, 1)$ are monoids. K is called a *strong bimonoid* if $\oplus$ is commutative and $0 \otimes k = k \otimes 0 = 0$ for each $k \in K$. For example, $(\mathbb{N} \cup \{\infty\}, +, \min, 0, \infty)$ is a (strong) bimonoid, named the *tropical bimonoid*.

Now fix a semiring/bimonoid K and an alphabet Σ. A *weighted finite automaton* (WFA) over K and Σ is a quadruple $A = (S, s, F, \mu)$ where S is a finite set of states, s is the starting state, F is set of the final states, $\mu : \Sigma \to K^{S\times S}$ is the transition weight function and λ is the weight of entering the automaton. For $\mu(a)(s, s') = k$, we write $s \xrightarrow{a}_k s'$.

Let At be the set of complete non-quantitative tests and P be the set of complete non-quantitative assignments. Let Ω be the set of complete quantitative tests and Δ be the set of complete quantitative assignments.

A weighted NetKAT automata is a finite state weighted automaton $A = (S, s, F, \lambda, \mu)$ over a structure K and alphabet Σ. The inputs to the automaton are so called reduced strings introduced in [1, 15], which belong to the set $U = \mathsf{At} \cdot \Omega \cdot \mathsf{P} \cdot \Delta \cdot (\mathsf{dup} \cdot \mathsf{P} \cdot \Delta)^*$, i.e., the strings belonging to U are of the form:

$$\alpha\omega p_0\delta_0 \ \mathsf{dup}\ \mathsf{p}_1\delta_1 \ \mathsf{dup}\ \cdots\ \mathsf{dup}\ \mathsf{p}_\mathsf{n}\delta_\mathsf{n}$$

for some $n \geq 0$. Intuitively, μ attempts to consume $\alpha\omega p_0\delta_0$ dup from the front of the input string and move to a new state with a weight and the new state has the residual input string $\alpha_0\omega_0\ p_1\delta_1\ \mathsf{dup}\ \cdots\ \mathsf{dup}\ \mathsf{p}_\mathsf{n}\delta_\mathsf{n}$.

The following construction shows the equivalence between WNetKAT and weighted automata.

From WFA to WNetKAT. Let $A = (S, s, F, \lambda, \mu)$ be a weighted NetKAT automata over K and Σ. An accepting path in A $s \xrightarrow{r_1}_{\alpha_1\beta_1} s_1 \xrightarrow{r_2}_{\alpha_2\beta_2} s_2 \cdots \xrightarrow{r_n}_{\alpha_n\beta_n} s_n$ can be write as the following WNetKAT expression:

$$\alpha_1\omega_1 p_1\delta_1\ \mathsf{dup}\ p_2\delta_2\ \mathsf{dup}\ \cdots\ \mathsf{dup}\ p_n\delta_n\,,$$

where

1. $\omega_1 = \lambda$, $\delta_1 = \omega_1 \oplus r$ and $\delta_i = \delta_{i-1} \oplus r_i$ for $i = 2, ..., n$;
2. $p_i = p_{\beta_i}$ for $i = 1, ..., n$.

From WNetKAT to WFA. Let e be a weighted automata expression, then following [1, 15], we can define a set of reduced strings R which are semantically equivalent to e. We define a weighted NetKAT automata $A = (S, s, F, \lambda, \mu)$ over a structure K and alphabet Σ, where

- $s = R$ and $\Sigma = \mathsf{At} \times \mathsf{At}$.
- $\mu : \Sigma \to K^{S\times S}$ is defined as: $\mu(\alpha, \beta)(u_1, u_2) = r$ iff $u_2 = \{\beta\omega' x \mid \alpha\omega p\delta\ \mathsf{dup}\ x \in u_1\}$, where $\beta = \alpha_p, \omega' = \delta_\omega$ and $\omega \otimes r = \omega'$. For short write $u_1 \xrightarrow{r}_{\alpha\beta} u_2$.
- $S = \{s\} \cup \{u \subseteq 2^U \mid \exists\ \mu\text{-path } s \to \cdots \to u\}$.
- $F = \{u \mid \alpha\omega p\delta \in u \in S\}$.
- $\lambda = \{\omega \mid \alpha\omega x \in s\}$.

We have the following theorem.

▶ **Theorem 6.**

1. *For every finite weighted WNetKAT automaton A, there exists a WNetKAT expression e such that the set of reduced strings accepted by A is the set of reduced strings of e.*
2. *For every WNetKAT expression e, there is a weighted WNetKAT automaton A accepting the set of the reduced strings of e.*

Let us just give some examples:

1. For the cost reachability use case, there exists a weighted WNetKAT automaton over the tropical semiring $(\mathbb{N} \cup \{\infty\}, +, \min, \infty, 0)$ that accepts the set of reduced strings of the WNetKAT expression in Section 4.1.

2. For the capacitated reachability: (i) There exists a weighted WNetKAT automaton over the semiring $(\mathbb{N} \cup \{\infty\}, \max, \min, 0, \infty)$ that accepts the set of the reduced strings of the WNetKAT expression for the splitable case in Section 4.2. (ii) There exists a weighted WNetKAT automaton over the tropical bimonoid $(\mathbb{N} \cup \{\infty\}, \min, +, 0, \infty)$ that accepts the set of the reduced strings of the WNetKAT expression for the unsplitable case in Section 4.2.

From this relationship, we have the following theorem about the (un)decidability of WNetKAT expression equivalence.

▶ **Theorem 7.** *Deciding equivalence of two WNetKAT expressions is equal to deciding the equivalence of the two corresponding weighted WNetKAT automata.*

For all the semiring and bimonoid we encountered in this paper, the WFA equivalence is undecidable. Therefore, the equivalence is also undeciable.

This negative result highlights the inherent challenges involved in complex network languages which are powerful enough to deal with weighted aspects.

However, we also observe that in many practical scenarios, the above undecidability result is too general and does not apply. For example, most of the use cases presented in in Section 4 can actually be reduced to test *emptiness*: we often want to test whether a given WNetKAT expression e equals 0, i.e., whether the corresponding weighted NetKAT automaton is empty. Indeed, there seems to exist an intriguing relationship between emptiness and reachability.

▶ **Theorem 8.** *Deciding whether a WNetKAT expression is equal to* 0 *is equal to deciding the emptiness of the corresponding weighted automaton.*

Interestingly, as shown in [7, 8, 21, 22], the emptiness problem is decidable for several semirings/bimonoids, e.g., the tropical semiring and the tropical bimonoid used in this paper. This leads to the decidability of the WNetKAT equivalence over these structures.

Another interesting domain with many decidability results are unambiguous regular grammars and unambiguous finite automata [40]. Accordingly, in our future work, we aim to extend these concepts to the weighted world and explore the unambiguous subsets of WNetKAT which might enable decidability for equivalence.

6 Conclusion

While OpenFlow today does not per se accommodate *stateful* packet operations or support arithmetic computations, we currently witness a trend toward computationally more advanced and stateful packet-processing functionality, see e.g., P4 or OpenState. Moreover, in order to implement arithmetic operations (see e.g., Equations (3) and (4)), we can simply use lookup tables realized as OpenFlow rules, see the technique in [32]. For a simple yet inefficient solution to compile *WNetKAT* switch variables is to use round robin groups [32].

In our future research, we aim to chart a more comprehensive landscape of the decidability and decision complexity of *WNetKAT* in different settings, and related to this, investigate the axiomatization in more depth. On the practical side, we are exploring possibilities for compiling *WNetKAT* to (extended) OpenFlow protocols such as P4 [4].

Finally, we refer the reader to our technical report [25] for additional details and use cases.

Acknowledgements. We would like to thank Alexandra Silva, Nate Foster, Dexter Kozen, Manfred Droste and Fredrik Dahlqvist for many inputs and discussions on WNetKAT.

References

1 Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.*, 49(1), January 2014. doi:10.1145/2578855.2535862.

2 Ryan Beckett, Michael Greenberg, and David Walker. Temporal NetKAT. In *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 386–401, 2016.

3 Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.*, 44(2), April 2014.

4 Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 44(3):87–95, 2014. doi:10.1145/2656877.2656890.

5 Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in active networks. *Communications Magazine, IEEE*, 36(10):72–78, 1998.

6 Manfred Droste and Paul Gastin. Weighted automata and weighted logics. In *Proc. ICALP*, 2005.

7 Manfred Droste and Doreen Götze. The support of nested weighted automata. In *Proc. Workshop on Non-Classical Models for Automata and Applications – (NCMA)*, 2013.

8 Manfred Droste and Doreen Heusel. The supports of weighted unranked tree automata. *Fundam. Inform.*, 2015.

9 Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of weighted automata*. Springer Science & Business Media, 2009.

10 Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 217–232, 2016.

11 Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN. *Queue*, 11(12):20:20–20:40, December 2013.

12 Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM SIGCOMM*, pages 327–338, 2013.

13 Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 279–291, 2011.

14 Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *Proc. ESOP*, 2016.

15 Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *ACM SIGPLAN Notices*, 2015.

16 Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Stephen Stuart Jonathan Zolla, Urs Hölzle, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Comput. Commun. Rev.*, 43(4), 2013. doi:10.1145/2486001.2486019.

17 Wolfgang John, Konstantinos Pentikousis, George Agapiou, Eduardo Jacob, Mario Kind, Antonio Manzalini, Fulvio Risso, Dimitri Staessens, Rebecca Steinert, and Catalin Meirosu. Research directions in network service chaining. In *Proc. IEEE SDN for Future Networks and Services*, 2013. doi:10.1109/SDN4FNS.2013.6702549.

18 Garvit Juniwal, Nikolaj Bjorner, Ratul Mahajan, Sanjit Seshia, and George Varghese. Quantitative network analysis. *Technical Report*, 2016.

19 Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proc. USENIX NSDI*, 2012.

20 Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proc. USENIX NSDI*, 2013.

21 Daniel Kirsten. The support of a recognizable series over a zero-sum free, commutative semiring is recognizable. In *Developments in Language Theory, 13th International Conference, DLT 2009, Stuttgart, Germany, June 30 – July 3, 2009. Proceedings*, pages 326–333, 2009.

22 Daniel Kirsten and Karin Quaas. Recognizability of the support of recognizable series over the semiring of the integers is undecidable. *Inf. Process. Lett.*, 111(10):500–502, 2011.

23 Turgay Korkmaz and Marwan Krunz. Multi-constrained optimal path selection. In *Proc. IEEE INFOCOM 2001*, volume 2, pages 834–843, 2001.

24 Dexter Kozen. *Kleene algebra with tests and commutativity conditions*. Springer, 1996.

25 Kim G. Larsen, Stefan Schmid, and Bingtian Xue. WNetKAT: A Weighted SDN Programming and Verification Language. In *ArXiv technical report 1608.08483*, 2016.

26 Tamas Lukovszki and Stefan Schmid. Online admission control and embedding of service chains. In *Proc. SIROCCO*, 2015.

27 Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proc. ACM SIGCOMM*, 2011. doi:10.1145/2018436.2018470.

28 Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, 2012.

29 Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proc. USENIX NSDI*, pages 1–14, 2013.

30 Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. Decentralizing SDN policies. In *ACM SIGPLAN Notices*, 2015.

31 M. Paredes-Farrera, M. Fleury, and M. Ghanbari. Router response to traffic at a bottleneck link. In *Proc. TRIDENTCOM*, 2006.

32 Liron Schiff, Michael Borokhovich, and Stefan Schmid. Reclaiming the brain: Useful openflow functions in the data plane. In *Proc. ACM HotNets*, 2014.

33 Liron Schiff, Petr Kuznetsov, and Stefan Schmid. In-Band Synchronization for Distributed SDN Control Planes. *Proc. ACM SIGCOMM CCR*, 2016.

34 Cole Schlesinger, Hitesh Ballani, Thomas Karagiannis, and Dimitrios Vytiniotis. Quality of service abstractions for software-defined networks. *Technical Report*, 2016.

35 David Schneider. The microsecond market. In *Proc. IEEE Spectrum*, 2012.

36 Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM 2012*, 2012. doi:10.1145/2342356.2342359.

37 Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *Proc. ACM HotNets-XIII*, 2014.

38 Jonathan M. Smith and Scott M. Nettles. Active networking: one view of the past, present, and future. *Proc. IEEE Transactions on Systems, Man, and Cybernetics: Applications and Reviews*, 34(1):4–18, 2004.

39 Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proc. ACM CoNEXT*, pages 213–226, 2014.

40 Richard E. Stearns and Harry B. Hunt. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. In *Proc. 22nd Annual Symposium on Foundations of Computer Science (SFCS)*, 1981.

41 Renata Teixeira, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker. Characterizing and measuring path diversity of internet topologies. In *ACM SIGMETRICS PER*, 2003. doi:10.1145/885651.781069.

42 Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2016.

43 Andreas Voellmy, Ashish Agarwal, and Paul Hudak. Nettle: Functional reactive programming for openflow networks. Technical report, Yale University, 2010.

44 Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: simplifying SDN programming using algorithmic policies. In *SIGCOMM CCR*, 2013. doi:10.1145/2534169.2486030.

45 Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally verifiable networking. *Proc. ACM HotNets*, 2009.

46 Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proc. USENIX NSDI*, 2015.

Deadline-Budget constrained Scheduling Algorithm for Scientific Workflows in a Cloud Environment

Mozhgan Ghasemzadeh[1], Hamid Arabnejad[2], and Jorge G. Barbosa[3]

1 **LIACC, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal**
`mozhgan.ghasemzadeh@fe.up.pt`
2 **IC4, Dublin City University, Dublin, Ireland**
`hamid.arabnejad@dcu.ie`
3 **LIACC, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal**
`jbarbosa@fe.up.pt`

Abstract

Recently cloud computing has gained popularity among e-Science environments as a high performance computing platform. From the viewpoint of the system, applications can be submitted by users at any moment in time and with distinct QoS requirements. To achieve higher rates of successful applications attending to their QoS demands, an effective resource allocation (scheduling) strategy between workflow's tasks and available resources is required. Several algorithms have been proposed for QoS workflow scheduling, but most of them use search-based strategies that generally have a higher time complexity, making them less useful in realistic scenarios. In this paper, we present a heuristic scheduling algorithm with quadratic time complexity that considers two important constraints for QoS-based workflow scheduling, time and cost, named Deadline-Budget Workflow Scheduling (DBWS) for cloud environments. Performance evaluation of some well-known scientific workflows shows that the DBWS algorithm accomplishes both constraints with higher success rate in comparison to the current state-of-the-art heuristic-based approaches.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Resource management, QoS scheduling, scientific workflow applications, deadline-constrained, budget-constrained

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.19

1 Introduction

Cloud computing infrastructures are the new platforms for tackling the execution needs of large-scale applications. Cloud computing promises the important benefits such as providing nearly-unlimited computing resources to execute application's task, on-demand scaling and pay-per-use metered service. Computing resources (i.e. virtual machines (VMs)) are dynamically allocated to user tasks based on application requirements, and users just pay for what they use. Each large-scale workflow application contains several tasks. Generally, workflow application can be represented by a Directed Acyclic Graph (DAG) that includes independent tasks, which can be executed simultaneously, or dependent tasks which need to be executed in a given other. In order to meet user's application QoS parameters, we need to find an efficient schedule map to execute the application tasks on multiple resources.

The majority of studies about workflow scheduling focus on single workflow application scheduling. However, these approaches are not adequate for cloud infrastructures due to

two major features: pay-as-you-go pricing model and on-demand resource provisioning. For example, in [1, 20, 28, 31, 30, 5, 13] authors considered fixed number of resources to the whole life time of the workflow application. But in our work, resources can be acquired at any time and released when they are idle, which save the total charged cost. Further, other approaches such as in [1, 20, 28, 31, 30] did not consider the hourly charging billing model in the cost model or the data transfer time in total time reservation of the virtual machine, which affects the effectiveness of the algorithm. In cloud computing infrastructures, such as Amazon EC2[1], the charging policy is based on a hour billing model even if the whole last reservation interval is not used. In this case, time fractions produced by previous tasks can be used by later tasks to save total renting cost. On the other hand, the workflow scheduling problem becomes more challenging when we consider multiple QoS parameters. Many algorithms have been proposed for multi-objective scheduling, but in most of them, meta-heuristic methods or search-based strategies have been used to achieve good solutions. However, these methods based on meta-heuristics or search-based strategies usually need significantly high planning costs in terms of the time consumed to produce good results, which makes them less useful in real platforms that need to obtain map decisions on the fly.

In this paper, a low-time complexity heuristic, named Deadline-Budget Workflow Scheduling (DBWS), is proposed to schedule workflow applications on cloud infrastructures constrained to two QoS parameters, namely, time and cost. The objective of the proposed DBWS algorithm is to find a feasible schedule map that satisfies the user defined deadline and budget constraint values. To fulfill this objective, the proposed approach implements a mechanism to control the time and cost consumption by each task when producing a schedule solution. To the best of our knowledge, the algorithm proposed here is the first *low-time* complexity heuristic, for cloud computing environments, addressing two QoS parameters as constraints.

The contributions of this paper are:

- a review of multiple QoS parameter workflow scheduling on cloud computing environments;
- a new heuristic algorithm with quadratic complexity for workflow application scheduling, constrained to time and cost;
- extensive evaluation with results for real-world applications.

The remainder of the paper is organized as follows. After outlining the related work in Section 2, we introduce the application and infrastructure model in Section 3. Section 4 presents the proposed scheduling algorithm. Section 5 presents results, and Section 6 concludes the paper.

2 Related work

The primary goal of many scheduling algorithms on cloud computing systems has focused on reducing the execution time of workflow applications without considering other factors such as the monetary cost or deadline. In [2, 24] we can find a taxonomy of scheduling algorithms for cloud computing systems. Considering multi-objectives for scheduling, we classify scheduling algorithms into the following two main categories: single workflow and multiple workflow scheduling algorithms. As the scheduling constraints on this paper are time and cost, we only consider these two QoS parameters in our review of previous work. Nevertheless, there are other QoS parameters such as reliability or energy that are not

[1] http://aws.amazon.com/ec2

considered here. Also, once our target platform is a cloud computing system, works that were proposed for grid infrastructures are not considered in this review because of different assumptions in the cost model. In cloud pricing model, i.e. a hour billing model, if the whole of the time interval is not used, it is still charged. Therefore, the formula used for cost consumption in grid platforms cannot be used in the cloud model and we cannot compare grid scheduling approaches with cloud ones in terms of cost consumption.

2.1 Single Workflow Scheduling Algorithms

In this category, the scheduling algorithms aim to find a suitable schedule map between workflow's task and available resources in order to meet application objective function which could be to optimize or to be constrained to a single or to multiple QoS parameters. Our work is related to the strategies which consider time and cost as QoS parameters for workflow scheduling.

2.1.1 Cost-optimization, deadline-constraint

The deadline of a workflow is defined as the maximum finish time of its last task to be executed. Calheiros et al. [5] developed an algorithm that is a cost-minimizer and applies replication of tasks to increase the chance of meeting application deadlines. Sahniet et al. [20] proposed a dynamic cost-effective deadline-constrained heuristic algorithm, namely JIT-C, for scheduling a scientific workflow in a public Cloud. In addition to these heuristic-based scheduling strategies, several works [16, 6] were proposed with the same objectives that by using search-based or meta-heuristic methods aims to find good solutions.

2.1.2 Time-optimization, budget-constraint

Budget is defined as the maximum amount that a user wants to pay for executing a workflow application on computing resources. In [13, 28, 31], authors proposed heuristic-based scheduling algorithms to minimize end-to-end execution under user-specified financial cost constraint. Zeng et al.[30] proposed a security-aware and budget-aware workflow scheduling strategy (SABA) for reducing the total execution time while meet required level of security.

2.1.3 Time-optimization, cost-optimization

Most strategies in this class try to mange the trade-off between running time and cost in order to minimize both QoS parameter time and cost in the provided schedule map. Selvarani et al. [22] proposed a job scheduling algorithm for making efficient mapping of independent tasks to available resources in a cloud. Lee et al. [11] proposed critical-path-first scheduling (CPF) algorithm which uses methods of stretching and compacting the workflow to optimize time and cost. In [4] authors proposed three bi-criteria complementary approaches for scheduling workflows on distributed Cloud resources. The first two algorithms, namely cost-based and time-based approaches, aim to minimize a single objective function (execution cost or time) individually by using Pareto approach, while the third algorithm, namely cost-time-based approach, is based on the obtained solutions by the two first algorithms for selecting only the Pareto solutions. By using the concept of Pareto dominance, authors in [25] proposed an algorithm that minimize total execution time and cost by setting a cost-efficient factor that represents the user's preference for the execution time and the monetary cost. The similar technique was used in [29]. Don et al. [14] proposed a framework which provided the balance between the application schedule performance and mandatory cost on Cloud

resources. However, our problem is different from these approaches in that both time and cost are treated as constraints at the same time, whereas these works try to consider one variable as constraint and optimize the other one, or target to optimize both cost and time.

2.1.4 Deadline-constraint, budget-constraint

Poola et al. [18] proposed a robust heuristic algorithm for scheduling a workflow on cloud computing systems considering deadline or budget constraints. The algorithm uses a search-based strategy to reassign scheduled tasks to new resources in order to satisfy the workflow constraint values for time or cost parameters. In [19], Rahman et al. present an adaptive hybrid heuristic (AHH) for workflow scheduling in hybrid cloud environment.

Two major drawbacks of the previous research work is that: a) usually, in their approaches the pricing model is the pay-as-go model similar to grid infrastructures and did not consider the billing model used in commercial cloud platform, i.e. the hour model; b) a fixed number of resources is considered in the scheduling process; and c) there is no timestamp for release/acquire of each VM resource.

2.2 Multiple Workflow Scheduling Algorithms

In contrast to single workflow scheduling, multiple workflows scheduling has received less attention. Li et al. [12] proposed two level workflow scheduling: the macro multi-workflow scheduling and the micro single workflow scheduling. Workflows are classified into time-sensitive and cost-sensitive based on QoS demands, and different scheduling strategies are adopted in order to meet each QoS time and cost requirements for each workflow type. In [27], authors proposed the Maximize Throughput of Multi-DAG with Deadline (MTMD) algorithm for scheduling concurrent workflow applications in order to improve the ratio of DAGs which can be accomplished within their deadline. Sharif et al. [23] proposed two online multiple workflow scheduling, namely OMPHC-PCPR and OPHC-TR, in Hybrid Cloud Environments. The difference between the two proposed algorithms is the ranking methodology to prioritize tasks during resource allocation.

3 Scheduling background

In this section, we formally describe the QoS workflow scheduling problem on cloud computing infrastructures.

3.1 Application model

Scientific workflow applications are commonly represented by a Directed Acyclic Graph (DAG), a directed graph with no cycles. Formally, a workflow application is a DAG represented by a triple $G =< T, E, data >$, where $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of tasks and n denotes the number of tasks in the workflow application. The set of edges E represent their data dependencies. A dependency ensures that a child node cannot be executed before all its parent tasks finish successfully and transfer the required input data. Let $data$ be a $n \times n$ matrix of communication data, where $data(t_i, t_j)$ is the file size required to be transmitted before task t_j execution from task t_i. The $\overline{C}_{(t_i \to t_j)}$ represents the average transfer time between the tasks t_i and t_j which is calculated based on the average bandwidth and latency among all resources pair. In a given DAG, a task with no predecessors is called an *entry task* and a task with no successors is called an *exit task*. We assume that the DAG has exactly

Table 1 Performance and price of various Amazon EC2 instances.

Instance type	Mean performance [GFLOPS]	Price[$/h]	GFLOPs/$
`m1.small`	2.0	0.1	19.6
`m1.large`	7.1	0.4	17.9
`m1.xlarge`	11.4	0.8	14.2
`c1.medium`	3.9	0.2	19.6

one entry task t_{entry} and one exit task t_{exit}. If a DAG has multiple entry or exit tasks, a dummy entry or exit task with zero weight and zero communication edges is added to the graph.

3.2 Resource model

The target cloud computing platform is composed of a set of m heterogeneous resources $R = \{\cup_{j=1}^{m} r_j \mid r_j \in VM_{type}\}$, that provide services of different capabilities and costs. Each resource includes computation service, e.g. Amazon Elastic Cloud Compute (EC2)[2], and storage service, e.g. Amazon Elastic Block Store (EBS)[3], used as a local storage device for saving the input/output files. In this study, all computation and storage resources are assumed to be in the same data center or region so that average bandwidth between computation resources is considered equal. Notice that the transfer time between two tasks being executed on the same VM is 0. Also, resources are offered in form of different type of virtual machines (VM_{type}). Each VM type has its own configuration for CPU capacity, memory size and an associated cost. Further, it is assumed that there is no limitation of the number of resources (VMs) used by a workflow application, and leasing a VM requires an initial boot time in order to be properly initialized and made available to the user; this time is not negligible and needs to be considered on the scheduling plan [15]. Similarly, on current commercial clouds, the pricing model is based on pay-as-you-go billing model for the number of *time intervals* used by a VM and it is specified by the cloud provider. The user will be charged for each complete *time interval* even if it does not completely use the time interval.

In this study, each resource r_j can be of any type as provided by Amazon EC2 (e.g. `m1.small`, `m1.large`, `m1.xlarge` and `c1.medium`). For a given resource r_j of a certain instance type, the average performance measured in GFLOPs and its price per hour of computation are known. The average performance in GFLOPs of four different Amazon EC2 instance types thorough extensive benchmark experimentation are evaluated in [8]. In our model, we assume that a task executed in any of these resources can benefit from a parallel execution using all the virtual cores exposed by the instance [7]. Also, according to Amazon cloud provider, users are charged based on the time interval of one hour ($interval\ time = 3600\,s$). Table 1 summarises the mean performance, the cost per hour of computation ($Cost_{r_j}$), and the ratio GFLOPs per invested dollar of these resources. Since all resource are located in the same data center or region, the internal data transfer cost is assumed to be zero.

[2] `http://aws.amazon.com/EC2/`
[3] `http://aws.amazon.com/EBS/`

Unlike previous work presented in Section 2, here, we propose an array of release/acquire ($VM_{r/a}$) timestamp for each VM resource which will be updated during the scheduling process. The array of timestamps $VM_{r/a} = \{(S_1, F_1), (S_2, F_2), \ldots\}$ where each pair (S, F) represents the Start and Finish time of consecutively execution of the target VM. These timestamps are calculated based on assigned tasks to the target VM.

During the scheduling process and after making final decision of the appropriate resource (r_{sel}) for execution of the current task (t_{curr}), if the current task could not benefit from last executed task on r_{sel} to reduce its execution cost, i.e. using the remaining last interval from the last previous scheduled task on r_{sel} , the $VM_{r/a}$ of resource r_{sel} is updated in the way that: a) add the release time after execution of the last scheduled task ; b) add the start (acquire) time according to the start time of t_{curr}. Otherwise, the release time of resource r_{sel} will be updated according to the finish time of the current task. Obviously, each resource can be rented for as many times and hours as required for finishing all the tasks scheduled on it. Additionally, we keep the set of scheduled tasks on each resource r_j denoted as $sched_{r_j} = \{t_i \mid AR(t_i) = r_j\}$, where $AR(t_i)$ represents the Assigned Resource on which the task t_i is scheduled to be executed. Each set $sched_{r_j}$ is sorted based on the finish time of its tasks.

3.3 Problem definition

The scheduling problem is defined as finding a map between tasks and resources in order to meet the QoS parameters defined for each job. The problem here consists in finding a schedule map in such a way that the total execution time (makespan) and economical costs are constrained to user's defined values for time and cost. We describe next how the two measures are computed.

3.3.1 Makespan

For computing the total execution time (makespan) of a given workflow, it is necessary to defined the *Time Reservation* (TR) of execution for task t_i on resource r_j as the sum of the execution time of task t_i on resource r_j ($ET(t_i, r_j)$) and the time required for transferring the biggest input data from any parent $t_p \in pred(t_i)$. The information of task execution time (ET) can be gathered via benchmarking or via precise performance models based on existing estimation techniques (e.g. historical data [9] and analytical modelling [17]).

$$TR(t_i, r_j) = \max_{t_p \in pred(t_i)} \left\{ \overline{C}_{t_p \to t_i} \right\} + ET(t_i, r_j)\,. \tag{1}$$

Considering the existence of data transfer time between tasks, for each task t_i to be executed in resource r_j, the resource r_j needs to be deployed before the task t_i starts transferring data from its parent and can only be released after its execution is finished and the data is transferred to its child task. First, we define $avail(r_j)$ as the earliest start Time of task t_i on resource r_j without considering its parents:

$$avail(r_j) = \begin{cases} 0 & , sched_{r_j} = \emptyset \\ FT(t_l, r_j) & , sched_{r_j} \neq \emptyset \end{cases} \tag{2}$$

where t_l is the last task in the sorted tasks scheduled list for resource r_j ($sched_{r_j}$) and $FT(t_l, r_j)$ in the Finish Time of t_l on r_j. Based on $avail(r_j)$, we defined the Release Time of resource r_j ($R_T(r_j)$) as the last rental period of one hour for the last scheduled task on it.

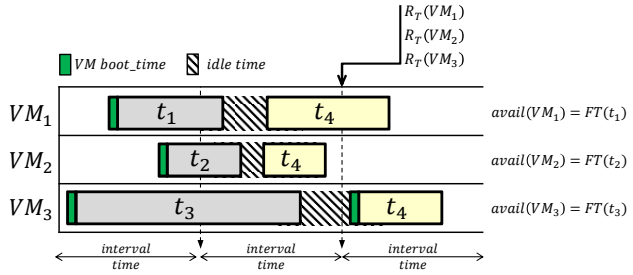


Figure 1 Example of a schedule. Tasks t_1, t_2 and t_3 are scheduled; and task t_4 is evaluated for scheduling.

After that, resource r_j will be released if no other task starts executing on the resource.

$$R_T(r_j) = \left\lceil \frac{avail(r_j)}{interval\ time} \right\rceil \times interval\ time\,. \tag{3}$$

Figure 1 shows a sample schedule generated for scheduled tasks t_1, t_2 and t_3 and current task t_4 which is selected to be scheduled. The Release Time of three resources, calculated by Eq. 3, are indicated as the last interval used by their last scheduled task. Note that, task t_4 is not scheduled and assigned to its target resource yet.

Next, the Start Time (ST) and Finish time (FT) of task t_i on each resource r_j are calculated as:

$$ST(t_i, r_j) = \max\left\{ \max_{t_p \in pred(t_i)} \{FT(t_p)\}, avail(r_j) \right\}, \tag{4}$$

$$FT(t_i, r_j) = \lambda_{(t_i,r_j)} + ST(t_i, r_j) + TR(t_i, r_j) \tag{5}$$

where $\lambda_{(t_i,r_j)}$ is defined as required boot time for acquiring resource instance r_j. If task t_i can be started at last interval time for resource r_j, no boot time required to be considered for task's completion, otherwise, the target resource r_j need to be lunched and its boot time should be considered as a delay in task finish time. For example, in Figure 1, only if task t_4 is scheduled on resource VM_3, a boot time is required to be consider on its finish time because it starts after current release time of VM_3. The $\lambda_{(t_i,r_j)}$ is calculated by:

$$\lambda_{(t_i,r_j)} = \begin{cases} 0 & ,ST(t_i, r_j) < R_T(r_j) \quad \text{OR} \quad sched_{r_j} = \emptyset \\ boot\_time_{r_j} & ,\text{otherwise} \end{cases} \tag{6}$$

where $boot\_time_{r_j}$ is the VM startup/boot time. In this study, we consider $boot\_time_{r_j} = 97\,s$ based on the measurements reported in [15] for the Amazon EC2 cloud. Please note that, during the resource selection phase for each task t_i, the $\lambda_{(t_i,r_j)}$ value is calculated according to the current situation of the target resources r_j, i.e. previous scheduled task on it.

The *makespan* or Schedule length is finally defined as the finish time of the last task of the workflow:

$$\text{DAG}_{makespan} = FT(t_{exit})\,. \tag{7}$$

3.3.2 Financial cost

The financial cost of task t_i on resource r_j is calculated based on the total usage time for complete task execution, considering data transfer time and execution time, and resource

usage price. In this research, we consider Amazon EC2 instance as the platform which makes hour price billing for each instance. In this model, partial hours are rounded up. As a consequence, if other tasks can be executed during that paid interval, they will not be charged for it. We define total usage time of task t_i as the payable period to be charged.

$$pay_{time}(t_i, r_j) = \begin{cases} FT(t_i, r_j) - ST(t_i, r_j) & , R_T(r_j) < ST(t_i, r_j) \\ 0 & , FT(t_i, r_j) < R_T(r_j) \\ FT(t_i, r_j) - R_T(r_j) & , \text{otherwise} \end{cases} \tag{8}$$

The *otherwise* condition will be applied if the task t_i starts before the current release time of resource r_j ($R_T(r_j)$) and finishes after it. So, in this case, the time slice before $R_T(r_j)$ is paid by previous tasks and should not be considered in the current usage time of task t_i.

The pay_{time} equal to zero means that the task can be executed on a previously paid interval (but not fully used) without any additional charge. For example, in Figure 1, if current task t_4 is scheduled on resource VM_2, the $pay_{time}(t_4, VM_2) = 0$. By considering Eq. 8, the pay time for resource VM_1 is equal to $pay_{time}(t_4, VM_1) = FT(t_4, VM_1) - R_T(VM_1)$ and for resource VM_3 we have $pay_{time}(t_4, VM_3) = FT(t_4, VM_3) - ST(t_4, VM_3)$. Please not that, the *boot_time* of resource VM_3 is already considered in $FT(t_4, VM_3)$ by Eq. 5.

The execution cost of task t_i for $pay_{time} > 0$ on resource r_j is computed by:

$$Cost(t_i, r_j) = \left\lceil \frac{pay_{time}(t_i, r_j)}{interval\ time} \right\rceil \times P_{r_j} \tag{9}$$

where P_{r_j} is the associated cost of resource r_j for each usage interval (Table 1, price column). Thus, the overall cost for executing a workflow application is:

$$\text{DAG}_{cost} = \sum_{t_i \in T} \left\{ Cost(t_i, r') \mid t_i \in sched_{r'} \right\} \tag{10}$$

4 Proposed Deadline–Budget workflow Scheduling (DBWS) algorithm

In this section, we present the Deadline-Budget Workflow Scheduling (DBWS) for cloud environments, which aims to find a feasible schedule within a budget and deadline constraints. The DBWS algorithm is a heuristic strategy that in a single step obtains a schedule that always accomplishes the deadline constraint and that may accomplish or not the budget constraint. If the cost constraint is met, we have a successful schedule, otherwise we have a failure and no schedule is produced. The algorithm is evaluated based on the success rate.

Before the description of the DBWS algorithm, next we present the attributes used in the algorithm.

- t_{curr} denotes the current task to be schedule, selected on the task selection phase among all ready tasks;
- r_{sel} denotes the target resource to execute t_{curr} on it;
- $FT_{min}(t_{curr})$ and $FT_{max}(t_{curr})$ denote the minimum and maximum finish time of current task among all tested resources;
- $\ell(t_i)$ denotes the level of task t_i; it is an integer value representing the maximum number of edges of the paths from the entry node to t_i. For the entry node, the level is $\ell(t_{entry}) = 1$, and for other tasks, it is given by:

$$\ell(t_i) = 1 + \max_{t_p \in pred(t_i)} \left\{ \ell(t_p) \right\}. \tag{11}$$

- $Cost_{min}(t_{curr})$ and $Cost_{max}(t_{curr})$ denote the minimum and maximum execution cost of the current task among all tested resources;
- $Cost_{high}(DAG)$ and $Cost_{low}(DAG)$ represent the total execution cost for scheduling target application workflow on the set of homogeneous VMs with highest and lowest cost among all possible VM types in our platform. Here, we use PEFT [3] algorithm to schedule a workflow application.

The DBWS algorithm consists of two phases, namely a *task selection* phase and a *resource selection* phase as described next.

4.1 Task selection

Tasks are selected according to their priorities. To assign a priority to a task in the DAG, the upward rank ($rank_u$) [26] is computed. This rank represents, for a task t_i, the length of the longest path from task t_i to the exit node(t_{exit}), including the computational time of t_i, and it is given by Eq. 12:

$$rank_u(t_i) = \overline{ET}(t_i) + \max_{t_{child} \in succ(t_i)} \left\{ \overline{C}_{t_i \to t_{child}} + rank_u(t_{child}) \right\} \tag{12}$$

where $\overline{ET}(t_i)$ is the average execution time of task t_i over all resources, $\overline{C}_{t_i \to t_{child}}$ is the average communication time between two tasks t_i and t_{child}, and $succ(t_i)$ are the set of immediate successor tasks of task t_i. To prioritize tasks it is common to consider average values because they have to be prioritize before knowing the location where they will run. For the exit node, $rank_u(t_{exit}) = \overline{ET}(t_{exit})$.

4.2 Resource Selection

The target VM to be selected to execute the current task is guided by the following quantities related to cost and time. To select the best suitable resource, a trade-off between these two variables, time and cost, is evaluated. We define a variable, S_{DL} as limit for time. S_{DL} is defined as the sub-deadline that is assigned to each task based on total application deadline. First, all tasks are divided in different levels based on their depth in the graph. We defined level execution ($Level_{exe}$) as the maximum execution length of all tasks in corresponding level and is given by:

$$Level^j_{exe} = \max_{\substack{t_i \in T \\ \ell(t_i) == j}} \left\{ ET_{max}(t_i) + \max_{t_p \in pred(t_i)} \{\overline{C}_{t_p \to t_i})\} \right\} \tag{13}$$

where $ET_{max}(t_i)$ represents the maximum execution time for task t_i among all VM_{type}. In the next step, we distribute the user deadline (D_{user}) among all levels. The sub-deadline value for level j ($Level^j_{DL}$) is computed recursively by traversing the task graph downwards, starting from the first level, as shown below:

$$Level^j_{DL} = Level^{j-1}_{DL} + D_{user} \times \frac{Level^j_{exe}}{\sum_{1 \le j' \le \ell(t_{exit})} Level^{j'}_{exe}}. \tag{14}$$

For the first level ($Level^1_{DL}$), the first part of Eq. (14) is considered zero. Finally, all tasks belonging to the same level have the same sub-deadline.

$$S_{DL}(t_{curr}) = \left\{ Level^j_{DL} \mid \ell(t_i) == j \right\}. \tag{15}$$

Note that, the task's sub-deadline is a soft limit as in most deadline distribution strategies; if the scheduler cannot find a resource (VM) that satisfies the sub-deadline for the current task, the resource that can finish the current task at the earliest time may is selected.

The resource selection phase is based on the combination of the two QoS factors, time and cost, in order to obtain the best balance between time and cost minimum values. We define two relative quantities, namely, Time Quality ($Time_Q$) and Cost Quality ($Cost_Q$), for current task t_{curr} on each resource $r_j \in R \cup R'$, where R represents the set of resources (VM instances) used in previous steps of scheduling, and R' is defined as the set of one temporary resource from each available VM_{type}. At each step after selecting the suitable resource r_{sel} for task t_{curr}, R is updated by $R = \{R \cup r_{sel} \mid r_{sel} \notin R\}$.

Both time and cost quantities are shown in (16) and (17), respectively. Both quantities are normalized by their maximum values.

$$Time_Q(t_{curr}, r_j) = \frac{\xi \times S_{DL}(t_{curr}) - FT(t_{curr}, r_j)}{FT_{max}(t_{curr}) - FT_{min}(t_{curr})} \tag{16}$$

$$Cost_Q(t_{curr}, r_j) = \frac{Cost_{max}(t_{curr}) - Cost(t_{curr}, r_j)}{Cost_{max}(t_{curr}) - Cost_{min}(t_{curr})} \times \xi \tag{17}$$

where

$$\xi = \begin{cases} 1 & \text{if} \quad FT(t_{curr}, r_j) < S_{DL}(t_{curr}) \\ 0 & otherwise \end{cases} \tag{18}$$

$Time_Q$ measures how much closer to the task sub-deadline (S_{DL}) the finish time of current task on resource r_j is. The sub-deadline defines the maximum allowance of task completion time. Consequently, resources with higher $Time_Q$ values, i.e. larger distance between finish time and sub-deadline, have higher possibility to be selected. If the current task has higher finish time on resource r_j than its sub-deadline, $Time_Q$ assumes a negative value for r_j, reducing the possibility for this resource to be selected. Similarly, $Cost_Q$ measures how much less the actual cost on resource r_j is than the maximum execution cost.

In the case that none of the resources can guarantee the current task sub-deadline ($S_{DL}(t_{curr})$), $Cost_Q$ is zero for all of them, and $Time_Q$ for each resource r_j is a negative value that represents the relative finish time obtained with r_j, i.e. a lower finish time causes a lower negative value. And, the resource with higher $Time_Q$, i.e. close to zero, would be selected.

Finally, to select the most suitable resource for current task, the Quality measure (Q) for each resource r_j is computed as shown in Eq. (19):

$$Q(t_{curr}, r_j) \;=\; Time_Q(t_{curr}, r_j) \times (1 - C_F) \;+\; Cost_Q(t_{curr}, r_j) \times C_F \tag{19}$$

where Cost-efficient Factor (C_F) is the tradeoff factor and defined as:

$$C_F = \frac{Cost_{low}(DAG)}{B_{user}}\,. \tag{20}$$

Both $Time_Q$ and $Cost_Q$ parameters are weighted by the ratio of the cheapest cost execution of the whole workflow application ($Cost_{low}(DAG)$) over the user defined available budget (B_{user}), so that the effectiveness of both time and cost factors can be controlled. A lower value of C_F means that the user prefers to pay more to execute the application faster, so that the time quality ($Time_Q$) is more predominant in the resource Quality measure (Q). In the same way, a higher value of C_F means that the user available budget is close

Algorithm 1 DBWS algorithm.

Require: a DAG and user's QoS Parameters values for Deadline (D_{user}) and Budget (B_{user})
1: Compute and sort all tasks based on their upward rank ($rank_u$) value
2: Compute PEFT schedule cost on the resources with cheapest ($Cost_{low}$) and most expensive ($Cost_{high}$) cost from VM_{type}
3: **if** $B_{user} < Cost_{low}(DAG)$ **then**
4: **return** no possible schedule map
5: **else if** $B_{user} > Cost_{high}(DAG)$ **then**
6: **return** PEFT scheduled map on most expensive VM_{type}
7: **end if**
8: Compute the Sub-DeadLine value (S_{DL}) for each task
9: **while** there is an unscheduled task **do**
10: t_{curr} = the next ready task with highest $rank_u$ value
11: **for all** $r_j \in R \cup R'$ **do**
12: Calculate Quality measure $Q(t_{curr}, r_j)$ using Eq. 19
13: **end for**
14: r_{sel} = resource r_j with highest Quality measure (Q)
15: Assign current task t_{curr} to resource r_{sel}
16: Update $R = \{R \cup r_{sel} \mid r_{sel} \notin R\}$
17: Update $VM_{r/a}(r_{sel})$
18: **end while**
19: **return** Schedule Map

to cheapest possible execution cost of the workflow, so that the time quality ($Time_Q$) is inconspicuous while the cost quality ($Cost_Q$) becomes more influential, allowing the selection of more cheap resources that guarantee a lower execution cost for t_{curr}.

The DBWS algorithm is shown in Algorithm 1. After some initializations in lines 1–2, first, the possibility of finding a schedule map under a user defined budget is checked in line 3. Then, the sub-deadline value for each task is computed according Eq. 15 in line 8. The DBWS algorithm starts to map all tasks of the application (while looping in lines 9–18). At each step, on line 10, among all ready tasks, the task with highest priority ($rank_u$) is selected as the current task (t_{curr}). Then, in lines 11–13, the Quality measure for assigning t_{curr} to the resource r_j ($Q(t_{curr}, r_j)$) is calculated. Note that, first, the finish time (FT) and execution cost of the current task is calculated and then the quality measure for all resources is calculated. Next, the resource (r_{sel}) with the highest quality measure among all resources is selected (line 14). Finally, after assigning the current task to the resource, the release/acquire timestamp for the resource r_{sel} is updated as explained in subsection *resource model* in Section 3.

In terms of time complexity, DBWS requires the computation of the upward rank ($rank_u$) and Sub-DeadLines (S_{DL}) for each task that have complexity $O(n.p)$, where p is the number of available resources and n is the number of tasks in the workflow application. In the resource selection phase, to find and assign a suitable resource for the current task, the complexity is $O(n.p)$ for calculating ST and FT for the current task among all resources, plus $O(p)$ for calculating the Quality measure. The total time is $O(n.p + n(n.p + p))$, where the total algorithm complexity is of the order $O(n^2.p)$.

5 Experimental results

This section presents performance comparisons of the DBWS algorithm with four most recently published algorithms, Hybrid [25], MTCT (Min-min based time and cost tradeoff)

[29], CwFT (Cost with Finish Time-based) [14] scheduling algorithms and SABA (Security-Aware and Budget-Aware) [30]. We choose MTCT[29] for comparison because it outperforms LOSS algorithms [21] and IC-PCP[1].

5.1 Budget and deadline parameters

To evaluate the DBWS algorithm, the user budget (D_{user}) and deadline (B_{user}) parameters assume values in a range so that the constraints are feasible. To specify these parameters, boundary values are defined for each of them, by using PEFT scheduling algorithm. We calculate the total execution time (makespan) of the workflow scheduled on the set of homogeneous VM instances with highest and lowest associated cost as the minimum (min_D) and the maximum (max_D) deadline boundary value. In the same way, the corresponding execution costs are the maximum (max_B) and minimum (min_B) execution cost of the workflow application. With these highest and lowest bound values, we define for the current application a unique Deadline and Budget constraint, as described by Eqs. (21) and (22):

$$D_{user} = min_D + \alpha_D \times (max_D - min_D) \tag{21}$$

$$B_{user} = min_B + \alpha_B \times (max_B - min_B) \tag{22}$$

where the deadline parameter α_D and budget parameter α_B can be selected in the range of $[0 \dots 1]$. In this paper, to observe the ability of finding valid schedule maps, we selected a low set of values, $\{0.1, 0.3, 0.5\}$, for time and cost parameters (α_D and α_B) to test the performance of each algorithm on harder conditions. Undoubtedly, increasing values for α_D and α_B, we would be able to achieve higher successful percentage rates.

5.2 Performance metric

To evaluate and compare our algorithm with other approaches, we consider the Planning Successful Rate (PSR), as expressed by Eq. (23). This metric provides the percentage of valid schedules obtained in a given experiment.

$$\text{PSR} = 100 \times \frac{\textit{Number of Successful Planning workflows}}{\textit{Total Number of workflows in experiment}}. \tag{23}$$

In addition, to investigate the quality of results, we compute the ratio of deadline defined and makespan achieved (NM) for each workflow, as well as the ratio of budget and execution cost (NB) of the schedule produced, as described in Eqs. (24) and (25):

$$\text{NM} = \frac{D_{user}}{\text{DAG}_{makespan}}, \tag{24}$$

$$\text{NB} = \frac{B_{user}}{\text{DAG}_{cost}}. \tag{25}$$

Note that, both metrics NM and NB are calculated for each schedule map, even for not successful ones, achieved by the algorithm. Basically, a lower value than 1 for NM and NB metrics means that the schedule map could not meet the constraint values for time and cost, respectively.

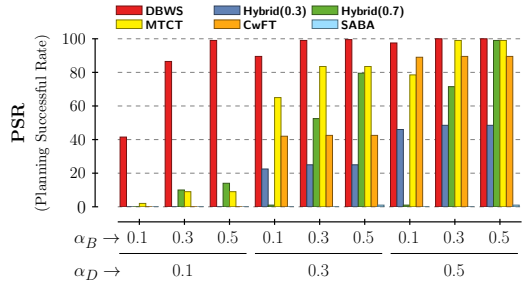


Figure 2 PSR value for CyberShake.

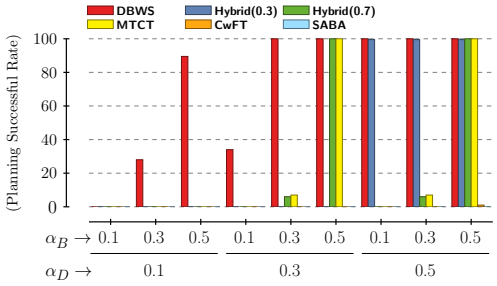


Figure 3 PSR value for Epigenomic.

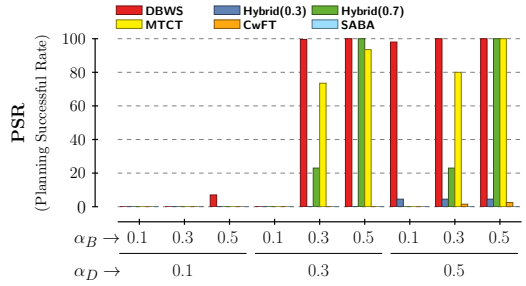


Figure 4 PSR value for Ligo.

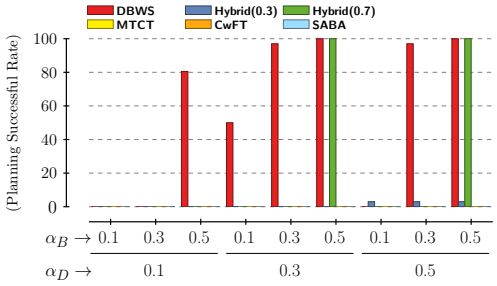


Figure 5 PSR value for Montage.

5.3 Results and discussion

To evaluate the algorithms on a standard and real set of workflow applications, a set of workflows were generated using the code developed in Pegasus toolkit[4]. Four well known structures were chosen [10], namely: CyberShake, Epigenomic, Ligo and Montage. The workflows are characterized as CPU-bound (Epigenomic), I/O-bound (Montage), data-intensive (CyberShake); workflows with large memory requirements (CyberShake, Ligo); and workflows with large resource requirements (CyberShake, Ligo). For each type of these real world workflows, we generated 1000 DAGs with a number of tasks equal to 50, 300 and 1000 tasks.

The original implementation of the compared scheduling algorithms assumed a fixed number of resources during the schedule map. In our implementation of those algorithms, we assigned an initialized fixed number of resources equal to the maximum number of concurrent tasks among all levels in the workflow application. Also, to apply the cost consumption during the scheduling process, we consider the same approach to calculate the cost execution of each task (Eq. 9) for all algorithms. For the Hybrid [25] scheduling algorithm, we consider two versions, namely Hybrid ($\alpha = 0.3$) and Hybrid($\alpha = 0.7$), where the α parameter represents the user's preference for minimizing the execution time or the monetary cost, i.e lower α corresponds to less monetary cost.

Figures 2, 3, 4 and 5 show the average Planning Successful Rate (PSR) obtained for the real workflow application considered here. The main result is that the algorithm DBWS obtains good performance in comparison to other state-of-the-art heuristic-based algorithms, for the range of budget and deadline values considered here. By increasing the budget factor (α_B), more budget is available to run the workflow resulting in an increase of the PSR value

[4] `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`

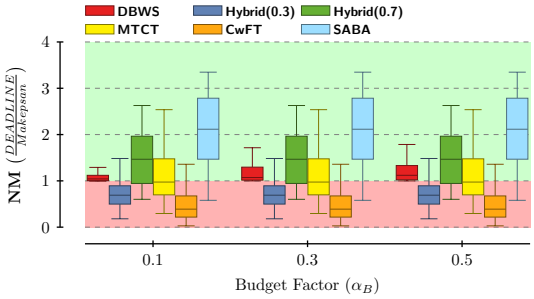


Figure 6 Deadline-makespan ratio.

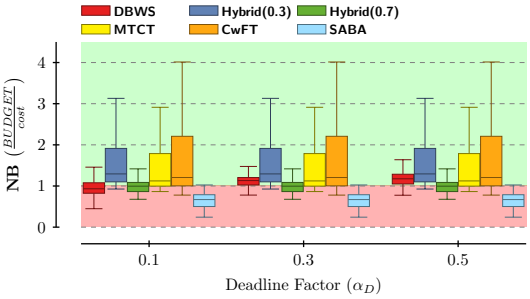


Figure 7 Budget-schedule cost ratio.

for DBWS. Note that for shorter deadlines ($\alpha_D = 0.1$) in the most cases only DBWS is able to complete workflows (Figures 3, 4 and 5).

Figures 6 and 7 represent the ratios related to time and cost, obtained by each algorithm. In other to have a better presentation, NM and NB values are divided into two main categories, where *safe* is represented by the green color and *unsafe* is represented by the red color. A value greater than 1 for the NM metric (Eq. 24) means that the algorithm could obtain a scheduled makespan lower than the user defined deadline. But a value $NM < 1$ means that the algorithm failed to find the schedule map with a makepsan lower than the user defined deadline. The same explanation can be considered for NB metric (Eq. 25). As it is shown in Figure 6, the makespan of the schedule map obtained by proposed DBWS algorithm always meet the user defined deadline constraint value for all range of budget factor α_B. However, for the total cost execution in Figure 7, by decreasing the deadline factor α_D, the execution cost of the schedule map obtained by DBWS becomes higher than the user defined budget. Note that the PSR values represented in Figures 2–5 represent the percentage of workflows for which the schedule meets both time and cost constraint values. For example, despite of the best reduction in execution cost by CwFT scheduling algorithm in Figure 7, due to failing in meeting the deadline value (Figure 6), the CwFT approach shows the worst performance in terms of PSR value (Figure 2–5). Also, for SABA scheduling algorithm, it fails in most cases as shown by the PSR metric. The reason can be explained due to the strategy used for VM assignment, namely the Comparative Factor (CF). The CF approach used the trade-off between time and cost factors and did not control the cost consumption during the scheduling process. As seen from Figures 6 and 7, SABA shows the best performance in total execution time reduction and worst one for the total cost.

6 Conclusions and future work

In this paper, we present the Deadline-Budget Workflow Scheduling (DBWS) algorithm for cloud environments, which maps a workflow application to cloud resources constrained to user-defined deadline and budget values. The algorithm was compared with other state-of-the-art heuristic-based scheduling algorithms. In terms of time complexity, which is a critical factor for effective usage on real platforms, our algorithm has quadratic time complexity. In terms of the quality of results, DBWS achieves better rates of successful schedules compared to other heuristic-based approaches for the real world applications considered. For the range values of deadline and budget constraints considered in this paper, DBWS shows a significant improvement of the planning successful rate for the workflows and cloud platform considered.

In conclusion, we have presented the DBWS algorithm for budget and deadline constrained scheduling, which has proved to achieve better performance than other heuristic-based approaches, namely Hybrid[25] MTCT[29] and CwFT[14].

In future work, we intend to extend the algorithm to consider dynamic concurrent workflow applications which can be submitted by any user at any time.

References

1 Saeid Abrishami, Mahmoud Naghibzadeh, and Dick H. J. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158–169, 2013.

2 Ehab Nabiel Alkhanak, Sai Peck Lee, and Saif Ur Rehman Khan. Cost-aware challenges for workflow scheduling approaches in cloud computing environments: Taxonomy and opportunities. *Future Generation Computer Systems*, 50(0):3–21, 2015.

3 Hamid Arabnejad and Jorge G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):682–694, 2014.

4 Kahina Bessai, Samir Youcef, Ammar Oulamara, Claude Godart, and Selmin Nurcan. Bi-criteria workflow tasks allocation and scheduling in cloud computing environments. In *5th Int. Conf. on Cloud Computing*, pages 638–645. IEEE, 2012.

5 Rodrigo N. Calheiros and Rajkumar Buyya. Meeting deadlines of scientific workflows in public clouds with tasks replication. *Parallel and Distributed Systems, IEEE Transactions on*, 25(7):1787–1796, 2014.

6 Wei-Neng Chen and Jun Zhang. A set-based discrete PSO for cloud workflow scheduling with user-defined QoS constraints. In *Int. Conf. on Systems, Man, and Cybernetics (SMC)*, pages 773–778. IEEE, 2012.

7 Juan J. Durillo and Radu Prodan. Multi-objective workflow scheduling in amazon ec2. *Cluster Computing*, 17(2):169–189, 2014.

8 Alexandru Iosup, Simon Ostermann, M. Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

9 S. Jang, Xingfu Wu, Valerie Taylor, Gaurang Mehta, Karan Vahi, and Ewa Deelman. Using performance prediction to allocate grid resources. Technical report, Technical Report 2004-25, GriPhyN Project, USA, 2004.

10 Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.

11 Young Choon Lee and Albert Y. Zomaya. Stretch out and compact: Workflow scheduling with resource abundance. In *13th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 219–226. IEEE, 2013.

12 Wenjuan Li, Qifei Zhang, Jiyi Wu, Jing Li, and Haili Zhao. Trust-based and qos demand clustering analysis customizable cloud workflow scheduling strategies. In *Cluster Computing Workshops*, pages 111–119. IEEE, 2012.

13 Xiangyu Lin and Chase Qishi Wu. On scientific workflow scheduling in clouds under budget constraint. In *42nd International Conference on Parallel Processing (ICPP)*, pages 90–99. IEEE, 2013.

14 Nguyen Doan Man and Eui-Nam Huh. Cost and efficiency-based scheduling on a general framework combining between cloud computing and local thick clients. In *Int. Conf. on Computing, Management and Telecommunications*, pages 258–263. IEEE, 2013.

15 Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *5th International Conference on Cloud Computing*, pages 423–430. IEEE, 2012.

16 Sahar Mirzayi and Vahid Rafe. A hybrid heuristic workflow scheduling algorithm for cloud computing environments. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(6):721–735, 2015. doi:10.1080/0952813X.2015.1020524.

17 Graham R. Nudd, Darren J. Kerbyson, Efstathios Papaefstathiou, Stewart C. Perry, John S. Harper, and Daniel V. Wilcox. Pace – a toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, 2000.

18 D. Poola, S. K. Garg, R. Buyya, Yun Yang, and K. Ramamohanarao. Robust scheduling of scientific workflows with deadline and budget constraints in clouds. In *28th Int. Conf. on Advanced Information Networking and Applications*, pages 858–865. IEEE, 2014.

19 Mustafizur Rahman, Xiaorong Li, and Henry Palit. Hybrid heuristic for scheduling data analytics workflow applications in hybrid cloud environment. In *Int. Symp. on Parallel and Distributed Processing (IPDPSW)*, pages 966–974. IEEE, 2011.

20 Jyoti Sahni and Deo Vidyarthi. A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2015.

21 Rizos Sakellariou, Henan Zhao, Eleni Tsiakkouri, and Marios D. Dikaiakos. Scheduling workflows with budget constraints. In *Integrated research in GRID computing*, pages 189–202. Springer, 2007.

22 S. Selvarani and G. Sudha Sadhasivam. Improved cost-based algorithm for task scheduling in cloud computing. In *Int. Conf. on Computational intelligence and computing research*, pages 1–5. IEEE, 2010.

23 Shaghayegh Sharif, Javid Taheri, Albert Y. Zomaya, and Surya Nepal. Online multiple workflow scheduling under privacy and deadline in hybrid cloud environment. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 455–462. IEEE, 2014.

24 Sucha Smanchat and Kanchana Viriyapant. Taxonomies of workflow scheduling problem and techniques in the cloud. *Future Generation Computer Systems*, 52:1–12, 2015.

25 Sen Su, Jian Li, Qingjia Huang, Xiao Huang, Kai Shuang, and Jie Wang. Cost-efficient task scheduling for executing large programs in the cloud. *Parallel Computing*, 39(4):177–188, 2013.

26 Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

27 Wei Wang, Qingbo Wu, Yusong Tan, and Fuhui Wu. Maximize throughput scheduling and cost-fairness optimization for multiple dags with deadline constraint. In *Algorithms and Architectures for Parallel Processing*, pages 621–634. Springer, 2015.

28 Chase Qishi Wu, Xiangyu Lin, Dantong Yu, Wei Xu, and Li Li. End-to-end delay minimization for scientific workflows in clouds under budget constraint. *Cloud Computing, IEEE Transactions on*, 3(2):169–181, 2015.

29 Heyang Xu, Bo Yang, Weiwei Qi, and Emmanuel Ahene. A multi-objective optimization approach to workflow scheduling in clouds considering fault recovery. *KSII Transactions on Internet and Information Systems*, 10(3):976–995, 2016.

30 Lingfang Zeng, Bharadwaj Veeravalli, and Xiaorong Li. Saba: A security-aware and budget-aware workflow scheduling strategy in clouds. *Journal of Parallel and Distributed Computing*, 75:141–151, 2015.

31 Lingfang Zeng, Bharadwaj Veeravalli, and Albert Y. Zomaya. An integrated task computation and data management scheduling strategy for workflow applications in cloud environments. *Journal of Network and Computer Applications*, 50:39–48, 2015.

Moving Participants Turtle Consensus*

Stavros Nikolaou[1] **and Robbert van Renesse**[2]

1 **Department of Computer Science, Cornell University, Ithaca, NY, US**
snikolaou@cs.cornell.edu
2 **Department of Computer Science, Cornell University, Ithaca, NY, US**
rvr@cs.cornell.edu

Abstract

We present Moving Participants Turtle Consensus (MPTC), an asynchronous consensus protocol for crash and Byzantine-tolerant distributed systems. MPTC uses various *moving target defense* strategies to tolerate certain Denial-of-Service (DoS) attacks issued by an adversary capable of compromising a bounded portion of the system. MPTC supports on the fly reconfiguration of the consensus strategy as well as of the processes executing this strategy when solving the problem of agreement. It uses existing cryptographic techniques to ensure that reconfiguration takes place in an unpredictable fashion thus eliminating the adversary's advantage on predicting protocol and execution-specific information that can be used against the protocol.

We implement MPTC as well as a State Machine Replication protocol and evaluate our design under different attack scenarios. Our evaluation shows that MPTC approximates best case scenario performance even under a well-coordinated DoS attack.

1998 ACM Subject Classification D.4.5 Fault-Tolerance

Keywords and phrases Consensus, adaptation, moving target defense

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.20

1 Introduction

Most distributed systems today are designed to tolerate failures. Existing fault-tolerance methods typically assume that failures are rare. They are tailored to provide good performance when no failures occur but might perform poorly under failure scenarios. However, as shown in works like [11], such designs allow malicious adversaries to craft workloads and Denial-of-Service (DoS) attacks that can substantially degrade the performance of certain state-of-the-art fault-tolerance protocols. As such DoS attacks become more common, it is becoming increasingly important to design fault-tolerance mechanisms that perform well in good scenarios while also gracefully handle adversarial ones. A core building block of many of these mechanisms are consensus protocols used by a set of replicas to agree on some state. One way to improve existing fault tolerance solutions is by enhancing the underlying consensus protocols with reconfiguration capabilities that allow them to change their execution parameters on the fly in order to better deal with adversarial workloads.

Our prior work on *Turtle Consensus* [22] also aims at attack-tolerant consensus. Turtle Consensus is a round-based consensus protocol that operates by using different consensus strategies across different rounds. The system's processes try to reach agreement running a

* This work was partially supported by AFOSR DURIP grant FA2386-12-1-3008, by NSF grants CCF-1047540, CNS-1040689, CNS-1422544, CNS-1561209, CNS-1601879, by a Google Faculty Research Award, by MDCN/iAd grant 54083, and by gifts from Infosys, Facebook, and Amazon.com. The authors would also like to thank the anonymous reviewers.

round of some consensus protocol in the literature; if they fail to do so they move onto the next round using a different protocol. The selection of each round's strategy is predetermined and known to all processes running the protocol. The main strategy of Turtle Consensus is to use the best approach available for normal operation in a particular setting and switch to different "backup protocols" as soon as the approach becomes inefficient, for example in the case of a DoS attack. We showed that the approach used sub-optimal strategies during DoS attacks, thus bounding the protocol's efficiency to the capabilities of these backup protocols. In addition, an adversary capable of compromising even a single consensus participant can learn and use the predetermined nature of the protocols' succession to constantly drive the system to sub-optimal executions.

In this paper we address these concerns by adding another degree of freedom in the reconfiguration capabilities of Turtle Consensus. We present Moving Participants Turtle Consensus (MPTC), an extension to the Turtle Consensus protocol that allows switching not only the protocols but also the set of processes on which they run across different rounds of a single consensus instance. The consensus protocol round and the processes participating in its execution form what we call a *configuration*, which our approach changes unpredictably at each round. While the configuration selection for each round is predetermined by a trusted dealer, it is unknown to the processes during MPTC execution. Using existing cryptographic techniques, we ensure that, only if sufficiently many processes collaborate during some round, the next round's configuration can be determined. This renders MPTC a valuable tool for building systems that can tolerate DoS attacks in crash-tolerant environments where a bounded portion of the system may be compromised. An extended version of this work that also handles Byzantine failures appears in [23].

2 Model

2.1 Processes and communication

Our system consists of a set of processes $\mathcal{N}$ that communicate using message passing. Each process is modeled as a state machine with a potentially unbounded set of states that executes transitions according to some protocol. The protocol specifies the transition function of the processes as well as the messages they exchange. Each process's state consists of two components, the *public* and the *private* or *secret state*. The public state contains the description of the protocol that each process executes and any public cryptographic keys associated with the process. The secret state contains any run-time state the process manages during the execution of the protocol as well as any secret cryptographic keys and/or shares associated with the process.

Protocol execution and communication are *asynchronous*, meaning that there are no bounds on the time it takes processes to execute transitions and deliver messages.

A process can be correct or faulty. A correct process faithfully executes the protocol and is guaranteed to make progress as long as the conditions specified by the protocol at any given step are eventually met. In this paper we only consider crash failures, although we extend our techniques to Byzantine failures in [23]. A faulty process may crash at any time after which it stops executing the protocol. Up to the point of the crash, processes faithfully follow the steps of the protocol. Communication between correct processes is reliable and secure. This means that, in the absence of a DoS attack (see below), messages sent by some correct process to another correct process are eventually delivered. It also means that messages are authenticated and cannot be tampered with or fabricated. We assume an upper bound, $f_c < |\mathcal{N}|$, on the total number of processes that might fail during the protocol's execution.

Finally, we assume the existence of a special process $T \notin \mathcal{N}$ that from now on we will refer to as *trusted dealer* or simply dealer. The dealer is only used during initialization of the system during which it generates the initial public and secret state of all processes. We assume that during this setup phase the dealer is correct, that it can communicate via private channels with any process in the system, and that it does not disclose its state. After initialization, however, the dealer does not execute any protocol steps or exchange messages with any other process, and the dealer's state is destroyed.

2.2 Adversary and attacks

We assume an adversary, A, that controls which processes fail and when. A is limited on the number of processes it can fail by f_c and cannot fail the dealer. A can also control the delivery order of messages of all processes as well as delay communication, but must yield to the previously stated reliable communication assumption.

The adversary can also issue *denial of service* (DoS) attacks against the system that can fully saturate the bandwidth resources of at most $f_a < |\mathcal{N}|$ correct processes. This can effectively prevent the targeted processes from progressing in the protocol's execution since they can no longer communicate with the rest of the system. A can change the targets of an attack over time and, in this way, can introduce communication and computation delays on certain processes. The adversary's objective is to prevent the system from making progress. From now on we will denote by f the maximum number of processes that can be crashed or under attack during the execution of the protocol, that is $f = f_c + f_a < |\mathcal{N}|$.

In this work, we ignore DoS attacks that target other resources like CPU using legitimate traffic. These attacks can be mitigated using rate limiting techniques such as client cryptographic puzzles [2].

The adversary has read access to the public state of all processes as well as the secret state of up to f processes. We call the processes whose secret state is disclosed to A *compromised*. While A cannot modify this state, it can use this state to select the target processes of a DoS attack. Once A has selected the set of compromised processes it can no longer change that set thus preventing A from accessing the secret state of more than f processes. Note that the set of compromised processes is not necessarily related to the set of processes that are crashed or under attack.

Finally, we assume that the various cryptographic schemes we are employing, like public key cryptography and threshold signatures, are secure in the random oracle model.

2.3 Cryptographic primitives

Our protocol relies on Threshold Coin Tossing [5]. Here we present a high-level description of this primitive that we will further formalize in Section 3. We employ an $(n, f + 1, f)$ threshold coin-tossing scheme in which n parties maintain shares of an unpredictable function, F, mapping an arbitrary bit string, r, to a binary value $\{0, 1\}$. Each of these shares can be used along with an input r to create a value that from now on we will refer to as *function shares*[1]. At least $f + 1$ of these function shares of r are required to reconstruct the result $F(r)$, while at most f parties may be compromised. We will use the term *function share of* $F(r)$ to denote a function share of r generated with a secret share of F.

The scheme defines three functions: 1) The *split* function, which takes as input a function F (represented as a bit string) and creates a set of shares as well as a verification key for

[1] The term used in [5] for these values is coin shares.

each of these shares. 2) The share combining function, *combine*, which takes an input r of F along with $f+1$ valid function shares of r and produces $F(r)$. 3) The share verification function *verify*, which takes an input r of F, a function share on r, and the verification key corresponding to the share that generated the input function share and determines whether the function share is valid.

This scheme is based on threshold signatures [26] and can be used to create an unpredictable sequence of bits while ensuring that it is computationally infeasible for the adversary to produce an input r and $f+1$ valid function shares that once combined do not yield $F(r)$. More formally, the scheme satisfies the following properties taken from [5]:

- *Robustness*: It is computationally infeasible for the adversary to produce a value r and $f+1$ valid shares of r such that the result of the combine function is not $F(r)$.
- *Unpredictability*: Given a value r and functions shares from fewer than $f+1-f$ correct processes, the adversary can predict the value of $F(r)$ with probability at most $\frac{1}{2}+\epsilon$ for negligible value $\epsilon \in \mathbb{R}$.

The previous unpredictability property was extended to sequences of output bits in [5], such that, given a sequence of values C_i for $i \in \{1, 2, \ldots, b\}$, an adversary with fewer than $f+1-f$ valid shares of some C_i has negligible advantage in predicting $F(C_i)$. From now on, when we talk about unpredictability we will refer to this *extended unpredictability property* of threshold coin-tossing.

Note that the previously described extended unpredictability property allows us to share unpredictable functions in $[\{0,1\}^* \rightarrow \{0,1\}^b]$ for any finite b. In other words, we can model each such function as a random number generator that can produce 2^b different values and requires $f+1$ processes to collaborate in order to produce the random (unpredictable) value corresponding to some arbitrary bit string r.

Threshold coin-tossing can be implemented using any non-interactive threshold signatures scheme that ensures unique valid signature per message as in [26]. A direct implementation of this scheme can be found in [5].

2.4 Underlying consensus protocols

MPTC, like other consensus protocols, solves the problem of agreement. In this problem, a set of possibly distributed processes, each of which is initialized with some input value, unanimously and irrevocably output one of those input values. More formally, let $\mathcal{N}$ be a set of processes each of which is initialized with some value from a value set $\mathcal{V}$. Each process can employ either of the following primitives:

- *propose* a value which allows a process to communicate its value to the rest of the processes in $\mathcal{N}$,
- *decide* a value which allows a process to output a value.

Every correct consensus protocol must satisfy the following properties:

- *Validity*: If a process decides a value, then that value must be the input value of some process in $\mathcal{N}$.
- *Agreement*: If any two processes decide they must decide the same value.
- *Termination*: All correct processes eventually decide.

[13] has shown that in an asynchronous environment no consensus protocol exists that satisfies all of the above properties when even only a single failure can occur. To circumvent this result, a variety of protocols have been proposed [3, 10] that use a probabilistic approach and can guarantee the previous properties with the following modification on termination:

All correct processes eventually decide with probability 1. For the remainder of this work we will refer to the non-probabilistic description of termination as *definite termination* and to the probabilistic one as *probabilistic termination*.

A consensus protocol that implements the previous specification (using either definite or probabilistic termination) even under the presence of f crash failures is called f-crash-resilient. Note that our adversary can additionally perform denial-of-service attacks which can fully saturate a bounded number of processes and render them entirely unavailable. In an asynchronous environment there is no difference between a crashed process and a process that is under DoS attack from the other processes' perspective. For this reason we say that a consensus protocol is correct in our model if it is f-crash-resilient where $f = f_c + f_a$. From now on we will refer to such consensus protocols as f-resilient protocols.

Each process executing MPTC may run different consensus protocols at different rounds. We denote the set of possible protocols each process can choose from by $\mathcal{P}$. Different consensus protocols make different assumptions under which they meet the previously described specification. The crash-tolerant consensus protocol of Ben-Or [3], for instance, assumes an asynchronous environment and that each infinite schedule has a bounded number of processes performing a finite number of steps. Other protocols make assumptions such as bounds on the number of failures, different degrees of synchrony, the existence of failure detectors [8], etc. We consider a consensus protocol correct if it satisfies agreement, validity and either definite or probabilistic termination. For each protocol $P \in \mathcal{P}$, we denote the set of assumptions required to hold for P to be correct by $\mathcal{A}_P$. In other words, if assumptions $\mathcal{A}_P$ hold, then P satisfies validity, agreement, and termination. A protocol P is a valid candidate for $\mathcal{P}$ if it is correct under both the assumptions in $\mathcal{A}_P$ and our previous model assumptions regarding failures, network reliability, and adversary.

We only consider consensus protocols operating in rounds and we follow the framework introduced in [22] for the specification of the round outcomes. According to this specification, every process running a round of a consensus protocol ends up in one of the following states: $\{D, U, M\} \times \mathcal{V}$, where states $(D, v), v \in \mathcal{V}$ indicate that the process has decided v, states $(U, v), v \in \mathcal{V}$ indicate that no process has decided up to the current round, and finally, states $(M, v), v \in \mathcal{V}$ indicate that while the process is not decided, if a decision was made by some process then it must have been v. We will refer to these states as round outcomes or simply *outcomes*. We denote by o_p^r the outcome of process $p \in \mathcal{N}$ at the end of round $r \in \mathbb{N}$.

More formally the following invariants hold about the outcomes of processes completing a round of a correct consensus protocol in $\mathcal{P}$:

▶ **Invariant 1.** *If* $\exists p \in \mathcal{N},\ r \in \mathbb{N}$ *such that* $o_p^r = (D, v)$*, where* $v \in \mathcal{V}$*, then for each correct* $q \neq p \in \mathcal{N}$ *it holds that* $o_q^r = (M, v)$ *or* $o_q^r = (D, v)$*.*

▶ **Invariant 2.** *If* $\exists p \in \mathcal{N}$*,* $r \in \mathbb{N}$ *such that* $o_q^r = (U, v)$ *for some* $v \in \mathcal{V}$ *then* $\forall q \in \mathcal{N}$*,* $u \in \mathcal{V}$*:* $o_q^r \neq (D, u)$*.*

This framework facilitates the description of MPTC in the next section and can be used to describe most consensus protocols in literature, including [3, 8, 18].

Problem. Our goal is to design a round-based consensus protocol that is correct under the previous system and adversary assumptions and that runs a different existing consensus protocol on a different set of processes each round. The selection of protocols and processes for each round must not be predictable by the adversary without the collaboration of correct processes. For the purposes of this work, unpredictability is as described in Section 2.3.

3 Moving Participants Turtle Consensus

In this section we describe our Moving Participants Turtle Consensus (MPTC) protocol. MPTC is an f-resilient consensus protocol operating in rounds such that in each round a different subset of processes may run a different consensus protocol. We start with some preliminary definitions and notation and then describe the protocol.

3.1 Participants and participant sets

MPTC is run by all processes in $\mathcal{N}$. In each round, only a subset of $\mathcal{N}$ is actively running a consensus protocol from a set of correct consensus protocols, $\mathcal{P}$. Let $\mathcal{P}_f$ correspond to the minimum number of processes required to run each protocol in $\mathcal{P}$. As an example, let $\mathcal{P}$ consist of the Ben-Or [3] and One-Third [9] consensus protocols. The first one requires $2f + 1$ processes to solve the agreement problem tolerating up to f crash failures while the second one needs $3f + 1$. Thus $\mathcal{P}_f = 3f + 1$. We assume that $|\mathcal{N}| \gg f$ and thus $|\mathcal{N}| > \mathcal{P}_f$ for most f-resilient consensus protocols.

In the remainder of this paper, we say that a process *runs* or *executes* a protocol in $\mathcal{P}$ when it executes a round of that protocol. We will refer to a process executing a protocol in $\mathcal{P}$ at some round of MPTC as a *participant* or an *active participant* of that round. Let $PS = \{S \subseteq \mathcal{N} : |S| = \mathcal{P}_f\}$ be the set of all possible subsets of $\mathcal{N}$ where each subset has size $\mathcal{P}_f$. We call each such set a *participant set*. A process may be a member of multiple participant sets. In each round r of MPTC, only a single participant set, S_r, is *active*, that is executing a consensus protocol in $\mathcal{P}$. We assume that participants in each participant set of some round r, $S_r \in PS$, are ordered and denote the i^{th} participant in S_r as S_r^i. The active participant set for each round is determined by T during initialization, which we describe later in this section.

3.2 Configurations

Before describing the initialization procedure and the core of MPTC, we need to define an important concept that encapsulates the information required for a set of processes to run a consensus protocol. We define a *configuration* of MPTC as a tuple $(P, S) \in \mathcal{P} \times PS$. $P \in \mathcal{P}$ describes the consensus protocol to run along with its initialization parameters. To better understand the information contained in the initialization parameters, consider a protocol like Lamport's Paxos [18] and the core consensus protocol he called Synod. In Synod, processes play multiple roles, such as proposers and acceptors. In that sense, P needs to encapsulate not only the protocol under execution, e.g. Synod, but also information related to its initialization such as mapping of proposers and acceptors to processes. The participant set $S \in PS$ corresponds to the set of processes that shall execute the consensus protocol specified by P. Let the set of all possible configurations $\mathcal{C} = \mathcal{P} \times PS$. Our approach implements a multi-party computation scheme for an unpredictable mapping between natural numbers (rounds) and configurations. We omit details regarding how to represent P since this is an implementation issue and does not affect our protocol. We assume that $|\mathcal{C}|$ is bounded.

3.3 Initialization and trusted dealer

We are now ready to describe the initialization of our protocol, how we are using T to create an unpredictable sequence of configurations, and how the active participants of a round can compute the corresponding configuration.

T is a special process that generates the configuration that each process in $\mathcal{N}$ starts with in the first round. It also provides the processes the means to generate configurations for subsequent rounds. To achieve this, T employs a $(\mathcal{P}_f, f+1, f)$ threshold coin tossing scheme like the one described in Section 2.3. Using this scheme, T shares a function F_S between the $\mathcal{P}_f$ processes of each participant set $S \in PS$. Recall that threshold coin-tossing can be implemented using threshold signatures, thus when we say that T shares a function F_S with each participant set, in reality it simply selects a different public-secret key pair for each $S \in PS$ and shares the secret key. Given some round number r, at least $f+1$ processes in S need to collaborate to produce $F_S(r)$ while up to f of them may get compromised. $f+1$ is both a sufficient and necessary number of processes to compute the result of the function shared. T cannot be compromised, failed or attacked by the adversary.

At a high level, T operates as follows:

1. For each $S \in PS$ the dealer picks a function $F_S : \{0,1\}^* \to \mathcal{C}$ and generates a secret share, h_S^q, for each $q \in S$.
2. T picks a configuration $C_0 \in \mathcal{C}$.
3. T distributes C_0 and shares to processes over private channels. $\forall S \in PS$ each process $p \in S$ receives h_S^p and C_0.

Observe that each function shared by the dealer maps arbitrary strings to configurations. This differs from the functions we defined in Section 2.3 which map arbitrary bit strings to bit strings of some finite length b. Since $\mathcal{C}$ is finite, there exists $b = \lceil log_2|\mathcal{C}| \rceil$ such that we can trivially obtain an onto function $\{0,1\}^b \to \mathcal{C}$. Thus, the functions we need to share can be trivially obtained by the ones supported by the threshold coin-tossing scheme. Note that, by this high-level algorithm, a process in $\mathcal{N}$ will receive multiple shares, one for each participant set it belongs to. The dealer selects each F_S such that the output is computationally indistinguishable from a randomly chosen function.

We now discuss how T generates the secret shares. Given model parameters $\mathcal{P}$ and f, T generates a different set of secret key shares for each subset, $S \in PS$. Each such set of secret key shares implicitly defines a function F_S mapping bit strings to configurations. We call this operation *split* and it is similar to the threshold coin-tossing dealer initialization described in [5]. *split* can be implemented using Shamir's secret sharing [25] $(\mathcal{P}_f, f+1)$. Note that the implementation in [5] is based on verifiable secret sharing because they are considering Byzantine failures. In our model, processes cannot lie and messages cannot be tampered with. As a result, no verification is needed within this context.

Given a secret share, h_S^p, of some function $F_S : \mathbb{N} \to \mathcal{C}$ and some input, $r \in \mathbb{N}$, process $p \in S$ can create a function share of $F_S(r)$ using the *share generation* function, $GFS : \mathcal{S} \times \mathbb{N} \to \mathcal{F}$ where $\mathcal{F}$ is the space of valid function shares that can be generated given a share $h \in \mathcal{S}$ and a natural number. We define, $F_S^p(r) = GFS(h_S^p, r)$. A straightforward implementation of GFS can be derived from the signature share generation for threshold signatures [26].

We define the *combine* functions as:

$$combine : \mathcal{F}^{f+1} \times \mathbb{N} \to \mathcal{C}$$

combine works by receiving function shares of some function F_S and some input, r and outputting $F_S(r)$ which corresponds to a configuration. More formally, let

$$F_S^Q(r) = \{F_S^q(r) \in \mathcal{F} \mid \forall q \in Q, Q \subseteq S \text{ and } |Q| = f+1\}$$

be any set of $f+1$ function shares of $F_S(r)$, i.e. $F_S^Q(r) \in \mathcal{F}^{f+1}$. Then we have that:

$$combine(F_S^Q(r), r) = F_S(r)$$

3.4 Protocol description

We can now describe the operation of each process executing MPTC. MPTC is an f-resilient round-based consensus protocol in which each round is executed under a different configuration. Let $C_r = (S_r, P_r) \in \mathcal{C}$ be the configuration used for round r, where $S_r \in PS$ is the active participant set and $P_r \in \mathcal{P}$ the consensus protocol specification for that round. Let o_r^p be the outcome of a process $p \in S_r$ running P_r at round r. Let a special value $\perp \notin \mathcal{V} \cup \mathcal{C} \cup \{\{D, M, U\} \times \mathcal{V}\}$ represent the value of an uninitialized variable.

We assume that all processes have common knowledge of $\mathcal{N}$, f, $\mathcal{P}$, $\mathcal{C}$ as well as of the functions *GFS combine*. Each process $p \in \mathcal{N}$ runs MPTC with its identifier and some value $x_p \in \mathcal{V}$ as input and at any point in time maintains the following state:

- its current round number, r_p, initialized to 0;
- its proposal $proposal_p$, initialized to x_p;
- the outcome of a round, o_p representing p's decision state and initialized to $\perp$ at the beginning of each round; and
- the current configuration c_p describing the currently known active participant set and the consensus protocol the active participants execute; it is initialized to C_0, which is provided by T during the initialization phase.
- the secret shares, h_S^p, $\forall S \in PS$ such that $p \in S$ provided by T during initialization.

We have organized MPTC description in phases. Messages exchanged between processes carry the number of the phase, the id of the sending process, and the current round along with the payload. Messages are of the form $\langle$*phase number*, *process id*, *round*, ...$\rangle$. Each round, r, of MPTC works in the following 3 phases:

- **Phase 1**: Each process $p \in S_r$ runs a round of the consensus protocol specified by C_r. Let o_p be p's outcome for round r. If $o_p = (D, v)$, then process p updates $proposal_p = v$, decides v and never updates o_p and $proposal_p$ again in any future round. If $o_p = (M, v)$, then p updates $proposal_p = v$. Regardless of o_p's value, p goes to Phase 2.
- **Phase 2**:
  - *Step 1*: Each $p \in S_r$ computes function share $F_{S_r}^p(r) = GFS(h_{S_r}^p, r)$ and sends a Phase 2 message $\langle 2, p, r_p, o_r^p, F_S^p(r)\rangle$ to all processes in S_r. Then p waits for Phase 2 messages from $\mathcal{P}_f - f$ processes in S_r. Once p receives enough messages from some $Q \subseteq S_r$, it proceeds to Step 2.
  - *Step 2*: If $o_p = (U, *)$ where $*$ can be any value in $\mathcal{V}$, then p updates its proposal to a value v, selected arbitrarily from the outcomes contained in the received Phase 2 messages. It also updates $o_p = (U, v)$.
  - *Step 3*: Let $F_S^Q(r)$ be the set of function shares received from processes in Q. p computes the configuration of the next round, $r + 1$, as $C_{r+1} = combine(F_{S_r}^Q(r), r)$ and moves on to Phase 3.
- **Phase 3**: Each $p \in S_r$ sends a Phase 3 message $\langle 3, p, r_p, o_p, C_{r+1}\rangle$ to each process in S_{r+1}. p updates its state: $r_p = r + 1$, $c_p = C_{r+1}$ and if it is still undecided, it updates its outcome, $o_p = \perp$. Each process $q \in S_{r+1}$ that receives Phase 3 messages with the same configuration value, C_{r+1}, from $\mathcal{P}_f - f$ processes, updates its proposal as follows. Let R denote the set of outcomes received:
  - Case 1: If $\exists o \in R$ such that $o = (D, v)$, then q updates $proposal_q = v$, decides v, sets its outcome $o_q = (D, v)$ and never updates o_q and $proposal_q$ again in any future round.
  - Case 2: If $\forall o \in R$ it holds $o = (M, v)$ for some $v \in \mathcal{V}$ then $proposal_q = v$.
  - Case 3: Otherwise, q selects an arbitrary outcome $(*, v) \in R$ where $*$ can be any value in $\{M, U\}$ and updates $proposal_q = v$.

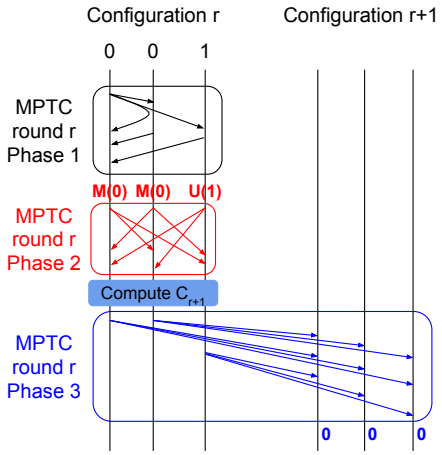


Figure 1 A round of MPTC consensus.

Then q sets $r_q = r + 1$, $c_q = C_{r+1}$ and if it is still undecided, it sets $o_q = \perp$. Finally, it starts the next round.

Figure 1 shows a visualization of the previous round description. MPTC runs for an unbounded number of rounds and eventually reaches a state in which a decision is made and all correct processes can eventually learn this decision. Messages from old rounds, either delayed in the network or sent by slow processes, are ignored while messages from future rounds are queued to be processed when the receiver reaches that round. The correctness of the previous protocol is presented in the full version of this paper in [23].

4 Implementation

In this section, we describe a simple implementation of MPTC as well as a state machine replication protocol we built on top of it. To implement MPTC we need to decide on the following parameters: the choice of protocol set $\mathcal{P}$, the set of possible configurations $\mathcal{C}$, the configuration selection functions F_S, $\forall S \in PS$, generated by the trusted dealer, and the implementations of *split*, *GFS* and *combine* functions.

Our set of protocols, $\mathcal{P}$, contains only a single consensus protocol, a parameterized version of single decree Paxos [18] in which each round comes with a predetermined leader known to all active participants. Paxos tolerates f crash failures using $2f + 1$ processes and under failure-free execution conditions, it can reach a decision within a single round-trip of communication. We assume the weakest failure detector, $\diamond\mathcal{W}$, presented in [7] which we implement using timeouts with exponentially increasing timeout periods. This way we ensure that there will be enough rounds executed by sufficiently many processes, which is critical for ensuring termination in our Paxos variant.

The timeouts mentioned above may cause certain processes executing our Paxos variant to exit a round without knowledge of the round's decision. Such processes need to retrieve this knowledge from the rest of the processes. To avoid incurring another round of communication in our Paxos variant, we piggyback this decision state retrieval onto Phase 2 of MPTC. Timed out processes can use the set of outcomes received to update their proposal.

Our set of configurations is $\mathcal{C} = \{(S, P) \mid S \in PS \text{ and } |S| = 2f + 1\}$ where $P \in \mathcal{P}$ is the described Paxos variant. Observe that in contrast to prior work on Turtle Consensus [22] we

use the same protocol across configurations. In Turtle Consensus, different configurations used the same $2f+1$ set of processes. As a result, the adversary could try to track the current leader within that set of processes even if the leader changed across different configurations. Therefore, a competent adversary could eventually locate and force Turtle Consensus rounds to fail, which can lead to poor performance. For that reason, Turtle Consensus kept switching between a leader-based (Paxos) and fully decentralized (Ben-Or) consensus protocols across configurations to prevent the adversary from exploiting the leader vulnerability. A side-effect of that approach, however, was that by falling back to a less efficient protocol (Ben-Or) it only achieved sub-par performance compared to the graceful execution using only Paxos rounds. With MPTC we do not need to employ such tactics since the adversary now needs to scan through $|\mathcal{N}| \gg f$ processes before it can identify the leader of our Paxos configuration.

In the implementation that we evaluate in Section 5 we did not implement the Threshold coin-tossing scheme. We emulated it instead by assuming that all participant sets use the same unpredictable function given to all processes via a configuration file. This file defines a sequence of configurations, one for each round, that processes move to in a round-robin fashion. We emulate the restrictions that the cryptographic framework imposes on the adversary by assuming that only the processes involved in rounds r and $r+1$ can learn C_{r+1} and only after Phase 2 of round r completes.

The interested reader can find an actual implementation of Threshold coin- tossing in [5]. In that work, they used cryptographically secure hash functions modeled as random oracles to implement unpredictable functions as well as for the *GFS* function. They also used Feldman's verifiable secret sharing [12] for *split* function, though in our crash-tolerant case Shamir's secret sharing [25] can be used instead. Finally, for *combine* they use Lagrange interpolation with coefficients the computed function shares.

For more details, see the full version of this work in [23].

4.1 MPTC-based state machine replication

We used the previous implementation of MPTC to build a SMRP, similar to the one described in [22]. While the components of the implementation are similar, their interactions are different. There are three sets of processes, the clients, the replicas $\mathcal{R}$, and the participants $\mathcal{N}$. The clients issue requests to the participants who order these requests and forward them to replicas. Replicas execute the received requests in the order established by participants and send the results back to participants who then forward them back to clients. Participants can additionally send reconfiguration messages to each other in order to update the configuration of the MPTC execution.

In greater detail, clients send uniquely identifiable requests to sufficiently many participants in order to ensure that at least one correct participant receives each request. The participants receive requests from clients and are responsible for ordering these requests and send them for execution to the replicas. Only one participant set can be active at any point in time. Any participant outside that set receiving a client request relays that request to the currently known active participant set. Active participants receiving client requests spawn MPTC instances, one for each request that needs to be ordered. Each instance has its own identifier and decided requests are ordered according to the identifiers of the MPTC instances that decided them. Clients can only communicate with participants and thus they are unable to launch DoS attacks on the replicas. MPTC is lazily instantiated for each slot and MPTC messages carry instance identifiers so incoming protocol messages are properly processed by the correct instance. If an instance has not yet been created, messages for that instance are queued and processed when it is created. Finally, there are at least $f+1$

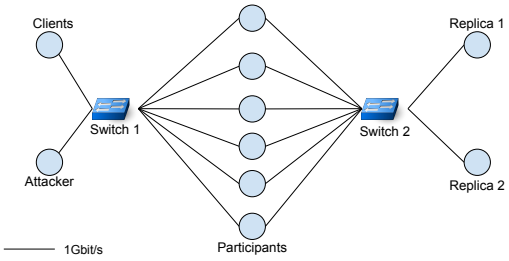


Figure 2 Experiment topology.

replicas, each of which maintains a copy of state of the service implemented by the SMRP. All replicas are initialized in the same state and execute the clients' requests in the order determined by id of the consensus instance created by the participants for each request.

For a detailed description of this SMRP implementation see [23].

5 Evaluation

In this section we present an evaluation of MPTC using the SMRP protocol presented in Section 4.1. In Section 5.1 we present the experiment setup and in Section 5.2 the performance results of MPTC under different attack scenarios.

5.1 Setup

We implemented MPTC and the SMRP described in Section 4.1 using `C++`. Our testbed consists of 10 nodes in Emulab [27], each with 8 cores running at 2.4 GHz, with 64GB of memory. For our experiments we used $f = 1$. Two nodes where designated as replicas, six as participants, one as clients, and one as the attacker. Nodes are connected by 1Gbps switched Ethernet as shown in Figure 2. Note that clients and attacker can only connect to participants, while participants connect to both replicas and clients. This choice was made to prevent the attacker from directly attacking the replicas of SMRP, thus degrading performance without attacking the consensus mechanism. All communication between participants takes place through Switch 1. Switch 2 is only used for participant to replica communication. We do not allow participants to communicate through Switch 2 since this would prevent the attacker from saturating the participants' bandwidth with respect to the MPTC execution. This would give MPTC an unfair advantage and would not showcase the benefits of its reconfiguration capabilities. All communication is over TCP/IP except for the DoS attack traffic, which is entirely UDP/IP. One of the two client nodes is used by the attacker and the other for creating legitimate client threads. We use a separate node for attacks in order to limit the effect of bandwidth attacks on the clients' ability to issue requests.

To simplify our evaluation, we set $\mathcal{C}$ to contain only two configurations such that the corresponding participant sets are disjoint. The configuration selection function provided by the trusted dealer (in our implementation by a configuration file) simply alternates between these two configurations every time a round fails. The predetermined Paxos leader of each configuration depends on the round in which the configuration is run and is rotated in a round-robin fashion every time the same participant set is reused. We consider that the attacker does not have this knowledge to make informed decisions regarding targeting processes.

Clients first connect to $f + 1$ random participants to which they issues requests. Once connected, each client executes the following loop: It issues each request to all $f + 1$ participants, waits for a response, discarding duplicate responses, and then sends the next request. Note that by connecting to $f + 1$ participants, we ensure that each client request reaches at least one correct participant who will further forward the request to the active participants. We have client requests contain no-ops, which means that when a decided request becomes ready for execution, replicas can immediately reply with a response.

The attacker creates a small number of attack threads, each of which targets a single participant, selects a random port, and sends UDP dummy messages as fast as it can. Note that these messages are not requests and are not processed by our participants since they never get to the application level. As in the Turtle Consensus evaluation [22], the goal of the attack is to prevent at most one participant from participating in MPTC instances. The attacker can focus all threads on the same participant or spread them across different ones. Since all attack threads are created on a single node, the aggregate bandwidth the attacker threads can saturate from the service cannot exceed 1Gbps.

We conducted experiments to test the throughput and latency of our implementation under normal execution and DoS attacks. Both metrics were measured at the client side. For throughput we measured the aggregate number of operations per second completed by client threads. Note that this is not the actual number of instances completed per second by our SMRP implementation since the same request might be decided more than once.

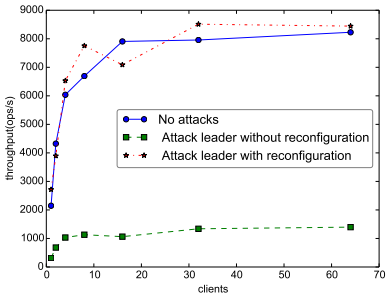
Other parameters of our experiment include:

- Duration: Each experiment lasted 1 minute. We found longer experiments did not significantly affect our metrics.
- Load: The number of concurrent clients, which ranged in our experiments from 1 to 64.
- Request size: The size of the command contained in each client request, which we set to 100 bytes.
- Attack message size: The size of the UDP messages send by attack threads to saturate the participants bandwidth; we set that to 1KB since our experimentation with our platform showed it is the smallest message size with the best results for the attacker.
- Number of attacker threads: Each run involving a DoS attack had 8 attack threads. We found that this number of threads yields best results for the attacker even when all 64 clients are connected to the target sharing the same link.
- Timeout: This is the initial timeout period used in our Paxos variant (Section 4) for each MPTC instance. Every time a round of some instance fails we double the timeout period for that instance.

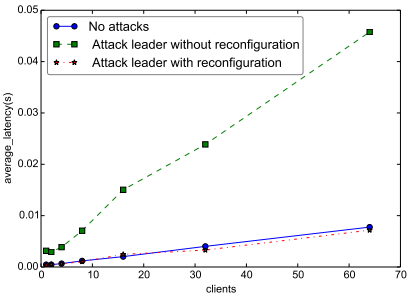
5.2 Results

In our evaluation, we investigated three main scenarios. In the first, we run our implementation of MPTC without any attacks taking place. The performance of this scenario will be our baseline since any attack scenarios drain resources from the system and thus is expected to perform similar or worse. This scenario is labeled “No attacks” in our figures.

The second scenario has the attacker focusing the DoS attack on a single node, the one that hosts the Paxos leader. This attack depletes the leader’s bandwidth. In this scenario no reconfiguration occurs. More specifically, we assume that in each round of MPTC the exact same configuration is chosen and the leader remains the same. Note that this scenario tries to simulate the case where the adversary can accurately track and attack the leader of the Paxos configuration. While any reasonable implementation of Paxos would change leaders



(a) Throughput as a function of load.



(b) Latency as a function of load.

Figure 3 Moving Participants Turtle Consensus performance under different attack scenarios.

among the $2f + 1$ processes, we set up the scenario to simplify issuing a very efficient attack. In our figures, this scenario is labeled "Attack leader without reconfiguration".

Finally, the third scenario uses an attacker who like in the previous scenario focuses on a single node. In this scenario the attacker is given the initial position of the leader but this time our implementation uses the MPTC version we described in Section 5.1 where consensus instances execution alternates between two disjoint sets of nodes. The attacker strategy here is to saturate the bandwidth of the known leader. It keeps attacking that node for the entirety of the experiment run. This attack is labeled "Attack leader with reconfiguration".

Figure 3a shows the throughput comparison of the previous three experiment scenarios as a function of the load on the SMRP. Each point represents the average throughput over 10 runs for each number of clients. In each of these runs clients connect to random participants, which in turn means that performance will vary across experiments. The first scenario is our best case scenario since the system operates at full resource capacity. The second scenario shows that performance suffers substantially when the Paxos leader is under attack. This is to be expected since the leader's participation is critical for making progress in each MPTC instance. In the third scenario we observe the benefits of the reconfigurable version of MPTC in action. The SMRP throughput is close to that of the No Attacks case. The main reason for this behavior is that since the leader of the first configuration is under attack and lacks the bandwidth to handle the valid traffic, some instance will inevitably fail the first round since the remaining participants will eventually time out. That will cause a reconfiguration that changes the active participant set. The new participants will pick up the failed instances as well as future requests and continue operating at full capacity. The minor deviations observed between scenarios 1 and 3 are mainly due to the randomness of client distribution over the set of all participants.

Figure 3b shows a comparison of the same scenarios as load increases, but this time with respect to latency. Observe that all scenarios behave similarly with latency linearly increasing with load. This behavior is to be expected since, as load increases, the number of concurrent MPTC instances increases, which in turn increases latency for each client. After all, each of them has to wait for a response to their previous request before sending the next one. As in the case of throughput, we see that both scenarios 1 and 3 have similar latencies while scenario 2 performs poorly. The reasoning is the same. In the second scenario the leader under attack is slower in completing instances, which raises the wait time for each client.

Note that this evaluation does not take into account the additional cost of reconfiguration that stems from the cryptographic operations required for threshold coin tossing like RSA

exponentiations. We therefore expect that under frequent reconfigurations there will be a wider gap between the performances of scenarios 1 and 3. However, we also expect that such reconfigurations will be infrequent, especially as the number of processes increases. Thus, while not an absolute comparison, our evaluation showcases the expected behavior and advantage of MPTC.

6 Related Work

A wide range of crash-tolerant consensus protocols have been proposed in literature each optimized for a different setting and/or metric. Some were designed to handle datacenter-scale systems like [6] which describes how Paxos was used to implement a fault-tolerant database for the Chubby locking service, an instance of which lies in each Google's datacenter. Others are focused on wide area deployments such as Mencius [21], which is a Paxos variant that employs multiple leaders each of which is responsible for a different set of consensus instances and may reside at different datacenters. Another important differentiating aspect of consensus protocols is whether they employ a special leader process like in [8, 18] or whether they are fully decentralized like the protocol proposed in [3]. This can greatly affect the behavior of a consensus protocol under different failure scenarios, including attacks, and was thus used by previous work on reconfigurable consensus [22] to design consensus protocols that provide acceptable performance under certain DoS attacks.

Our work resembles the work on Vertical Paxos [19]. Vertical Paxos is a reconfigurable state machine replication protocol that uses a special auxiliary master process to decide the next configuration of the system including the set of replicas participating in that configuration. Unlike Vertical Paxos, MPTC does not require additional online master processes to compute the next configuration. Our assumed trusted dealer is only active during initialization. In addition, Vertical Paxos is not designed for an adversary capable of compromising even a single process and thus would not perform as well against the DoS attacks described in this work.

Moving target defenses have often been used as response to DoS and Distributed DoS (DDoS) attacks. [14] proposes changing the IP address of the target node for dealing with local IP-based DoS attacks. More recently in [16], Software-Defined Networking (SDN) has been used to implement moving target defense approaches like "random host mutation" in which, similarly to [14], the controller periodically alters the virtual IP addresses of hosts to hide the real IP addresses from an intruder. Our Moving Participants Turtle Consensus approach resembles more the "proactive server roaming" approach in [17]. That is an adaptive approach in which the active server proactively switches servers from an existing pool in order to deal with unpredictable and undetectable attacks. Their approach ensures that only legitimate clients can track the moving server. Like in the case of our MPTC protocol, proactive server roaming performs gracefully during attacks. However, it imposes significant overhead in attack-free scenarios, which is not the case for MPTC since we only reactively change configurations.

Our work assumes an adversary that cannot change the set of corrupted processes over time. Other related work has focused on dynamic models of corruption. [15] introduced proactive secret sharing, an instance of proactive security [24] for supporting secure computation in synchronous distributed systems. These ideas have been adapted to asynchronous ones in [4, 28]. While these approaches did not consider DoS attacks, they are orthogonal to ours and can be used to further improve this work for dealing with mobile adversaries.

Running consensus on a subset of a larger set of processes to decrease message complexity has been explored in [1]. It has also been explored more recently in [20] for improving the scalability of Byzantine agreement on blockchains.

7 Conclusions

In this paper we presented Moving Participants Turtle Consensus (MPTC), an extension to the Turtle Consensus protocol [22] that allows running different consensus protocols, on different sets of processes, across different rounds of a single consensus instance. MPTC can deal with adversaries with bounded information on the system by making unpredictable changes in the execution of the protocol. Our evaluation of our prototype implementation of MPTC suggests that we can achieve the performance offered by the most efficient consensus protocols even when the system is under attack.

References

1 Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In Fabian Kuhn, editor, *Distributed Computing – 28th International Symposium, DISC 2014, Austin, TX, USA, October 12–15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2014.

2 Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In *Security Protocols*, volume 2133 of *Lecture Notes in Computer Science*, pages 170–177. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-44810-1_22.

3 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proc. of the 2nd Annual ACM Symp. on Principles of Distributed Computing*, PODC'83, pages 27–30, New York, NY, USA, 1983. ACM. doi:10.1145/800221.806707.

4 Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *in Proc. 9th ACM Conference on Computer and Communications Security (CCS*, pages 88–97. ACM Press, 2002.

5 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005. doi:10.1007/s00145-005-0318-0.

6 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, PODC'07, pages 398–407, New York, NY, USA, 2007. ACM. doi:10.1145/1281100.1281103.

7 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996. doi:10.1145/234533.234549.

8 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996. doi:10.1145/226643.226647.

9 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009. doi:10.1007/s00446-009-0084-6.

10 Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, August 1994. doi:10.1137/S0097539790192635.

11 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages

153–168, Berkeley, CA, USA, 2009. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1558977.1558988.

12 Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *The 28th Annual Symposium on Foundations of Computer Science*, pages 427–438, Oct 1987. doi:10.1109/SFCS.1987.4.

13 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

14 Xianjun Geng and Andrew B. Whinston. Defeating distributed denial of service attacks. *IT Professional*, 2(4):36–42, Jul 2000. doi:10.1109/6294.869381.

15 Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proc. of the 15th Annual Int. Cryptology Conf. on Advances in Cryptology*, CRYPTO'95, pages 339–352, London, UK, 1995. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646760.706016.

16 Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proc. of the 1st Workshop on Hot Topics in Software Defined Networks*, pages 127–132. ACM, 2012. doi:10.1145/2342441.2342467.

17 Sherif. M. Khattab, Chatree Sangpachatanaruk, Rami Melhem, Daniel Mosse, and Taieb Znati. Proactive server roaming for mitigating denial-of-service attacks. In *International Conference on Information Technology: Research and Education (ITRE 2003)*, pages 286–290, Aug 2003. doi:10.1109/ITRE.2003.1270623.

18 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

19 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*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM. doi:10.1145/1582716.1582783.

20 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*, CCS'16, pages 17–30, New York, NY, USA, 2016. ACM. doi:10.1145/2976749.2978389.

21 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855741.1855767.

22 Stavros Nikolaou and Robbert van Renesse. Turtle consensus: Moving target defense for consensus. In *Proceedings of the 16th Annual Middleware Conference*, Middleware'15, pages 185–196, New York, NY, USA, 2015. ACM. doi:10.1145/2814576.2814811.

23 Stavros Nikolaou and Robbert van Renesse. Moving Participants Turtle Consensus. Technical report, Cornell University, November 2016. arXiv:1611.03562.

24 Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'91, pages 51–59, New York, NY, USA, 1991. ACM. doi:10.1145/112600.112605.

25 Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

26 Victor Shoup. Practical threshold signatures. In *Proc. of the 19th Int. Conf. on Theory and Application of Cryptographic Techniques*, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1756169.1756190.

27 Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation (OSDI'02)*, pages 255–270, Boston, MA, December 2002. Usenix.

28 Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. APSS: Proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, August 2005. doi:10.1145/1085126.1085127.

Kleinberg's Grid Reloaded

Fabien Mathieu

Nokia Bell Labs, Nozay, France
fabien.mathieu@nokia-bell-labs.com

Abstract

One of the key features of small-worlds is the ability to route messages with few hops only using local knowledge of the topology. In 2000, Kleinberg proposed a model based on an augmented grid that asymptotically exhibits such property.

In this paper, we propose to revisit the original model from a simulation-based perspective. Our approach is fueled by a new algorithm that uses dynamic rejection sampling to draw augmenting links. The speed gain offered by the algorithm enables a detailed numerical evaluation. We show for example that in practice, the augmented scheme proposed by Kleinberg is more robust than predicted by the asymptotic behavior, even for very large finite grids. We also propose tighter bounds on the performance of Kleinberg's routing algorithm. At last, we show that fed with realistic parameters, the model gives results in line with real-life experiments.

1998 ACM Subject Classification C.2.2 Routing Protocols, C.2.4 Distributed Systems

Keywords and phrases Small-World Routing, Kleinberg's Grid, Simulation, Rejection Sampling

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.21

1 Introduction

In a prescient 1929 novel called *Láncszemek* (in English, *Chains*), Karinthy imagines that any two people can be connected by a small chain of personal links, using no more than *five* intermediaries [10].

Years later, Milgram validates the concept by conducting real-life experiments. He asks volunteers to transmit a letter to an acquaintance with the objective to reach a target destination across the United States [17, 20]. While not all messages arrive, successful attempts reach destination after six hops in average, popularizing the notion of *six degrees of separation*.

Yet, for a long time, no theoretical model could explain why and how this kind of *small-world* routing works. One of the first and most famous attempts to provide such a model is due to Jon Kleinberg [12]. He proposes to abstract the social network by a grid augmented with *shortcuts*. If the shortcuts follow a heavy tail distribution with a specific exponent, then a simple greedy routing can reach any destination in a short time ($O(\log^2(n))$ hops). On the other hand, if the exponent is wrong, then the time to reach destination becomes $\Omega(n^\alpha)$ for some α. This seminal work has led to multiple studies from both the theoretical and empirical social systems communities.

Contribution

In this paper, we propose a new way to numerically benchmark the greedy routing algorithm in the original model introduced by Kleinberg. Our approach uses dynamic rejection sampling, which gives a substantial speed improvement compared to previous attempts, without making any concession about the assumptions made in the original model, which is kept untouched.

Fueled by the capacity to obtain quick and accurate results even for very large grids, we give a fresh look on Kleinberg's grid, through three independent small studies. First, we show that the model is in practice more robust than expected: for grids of given size there is quite a large range of exponents that grant short routing paths. Then we observe that the lower bounds proposed by Kleinberg in [12] are not tight and suggest new bounds. Finally, we compare Kleinberg's grid to Milgram's experiment, and observe that when the grid parameters are correctly tuned, the performance of greedy routing is consistent with the *six degrees of separation* phenomenon.

Roadmap

Section 2 presents the original augmented grid model introduced by Kleinberg and the greedy routing algorithm. A brief overview of the main existing theoretical and experimental studies is provided, with a strong emphasis on the techniques that can be used for the numerical evaluation of Kleinberg's model.

In Section 3, we give our algorithm for estimating the performance of greedy routing. We explain the principle of dynamic rejection sampling and detail why it allows to perfectly emulate Kleinberg's grid with the same speed that can be achieved by toroidal approximations. We also give a performance evaluation of the simulator based on our solution. For readers interested in looking under the hood, a fully working code (written in Julia) is given in Appendix A.

To show the algorithm benefits, we propose in Section 4 three small studies that investigate Kleinberg's model from three distinct perspectives: robustness of greedy routing with respect to the shortcut distribution (Section 4.1); tightness of the existing theoretical bounds (Section 4.2); emulation of Milgram's experiment within Kleinberg's model (Section 4.3).

2 Model and Related Work

We present here the model and notation introduced by Kleinberg in [12, 11], some key results, and a brief overview of the subsequent work on the matter.

2.1 Kleinberg's Grid

In [12], Kleinberg considers a model of directed random graph $G(n, r, p, q)$, where n, p, q are positive integers and r is a non-negative real number. A graph instance is built from a square lattice of $n \times n$ nodes with Manhattan distance d: if $u = (i, j)$ and $v = (k, l)$, then $d(u, v) = |i - j| + |k - l|$. d represents some natural proximity (geographic, social, ...) between nodes. Each node has some *local* neighbors and q *long range* neighbors. The local neighbors of a node u are the nodes v such that $d(u, v) \leq p$. The q long range neighbors of u, also called *shortcuts*, are drawn independently and identically as follows: the probability that a given long edge starting from u arrives in v is proportional to $(d(u, v))^{-r}$.

The problem of decentralized routing in a $G(n, r, p, q)$ instance consists in delivering a message from node u to node v in a hop-by-hop basis. At each step, the message bearer needs to choose the next hop among its neighbors. The decision can only use the lattice coordinates of the neighbors and destination. The main example of decentralized algorithm is the *greedy routing*, where at each step, the current node chooses the neighbor that is closest to destination based on d (in case of ties, an arbitrary breaking rule is used).

The main metric to analyze the performance of a decentralized algorithm is the *expected delivery time*, which is the expected number of hops to transmit a message between two nodes chosen uniformly at random in the graph.

This paper focuses on studying the performance of the greedy algorithm. Unless stated otherwise, we assume $p = q = 1$ (each node has up to four local neighbors and one shortcut). Let $e_r(n)$ be the expected delivery time of the greedy algorithm in $G(n, r, 1, 1)$.

2.2 Theoretical Results

The main theoretical results for the genuine model are provided in the original papers [11, 12], where Kleinberg proves the following:

- $e_2(n) = O(\log^2(n))$;
- for $0 \leq r < 2$, the expected delivery time of any decentralized algorithm is $\Omega(n^{(2-r)/3})$;
- for $r > 2$, the expected delivery time of any decentralized algorithm is $\Omega(n^{(r-2)/(r-1)})$.

Kleinberg's results are often interpreted as follows: short paths are easy to find only in the case $r = 2$. The fact that only one value of r asymptotically works is sometimes seen as the sign that Kleinberg's model is not robust enough to explain the small-world routing proposed by Karinthy and experimented by Milgram. However, as briefly discussed by Kleinberg in [5], there is in fact some margin if one considers a grid of given n. This tolerance will be investigated in more details in Section 4.1.

While we focus here on the original model, let us give a brief, non-exhaustive, overview of the subsequent extensions that have been proposed since. Most proposals refine the model by considering other graph models or other decentralized routing algorithms. New graph models are for example variants of the original model (studying grid dimension or the number of shortcuts per node [2, 5, 9]), graphs inspired by peer-to-peer overlay networks [14], or arbitrary graphs augmented with shortcuts [6, 8]. Other proposals of routing algorithms usually try to enhance the performance of the greedy one by granting the current node additional knowledge of the topology [7, 14, 15].

A large part of the work above aims at improving the $O(\log^2(n))$ bound of the greedy routing. For example, in the small-world percolation model, a variant of Kleinberg's grid with $O(\log(n))$ shortcuts per node, greedy routing performs in $O(\log(n))$ [14].

2.3 Experimental Studies

Many empirical studies have been made to study how routing works in real-life social networks and the possible relation with Kleinberg's model (see for example [13, 5] and the references within). On the other hand, numerical evaluations of the theoretical models are more limited to the best of our knowledge. Such evaluations are usually performed by averaging R runs of the routing algorithm considered.

In [11], Kleinberg computes $e_r(n)$ for $n = 20,000$ and $r \in [0, 2.5]$, using 1,000 runs per estimate. However, he uses a torus instead of a regular grid (this will be discussed later in the paper). In [14], networks of size up to 2^{24} (corresponding to $n = 2^{12}$ in the grid) are investigated using 150 runs per estimate.

Closer to our work, Athanassopoulos *et al.* propose a study centered on numerical evaluation that looks on Kleinberg's model and some variants [1]. For the former, they differ from the original model by having fixed source and destination nodes. They compute $e_r(n)$ for values of n up to 3,000 and $r \in \{0, 1, 2, 3\}$, using 900 runs per estimate.

To compare with, in the present paper, we consider values of n up to $2^{24} \approx 16,000,000$ and $r \in [0, 3]$, with at least 10,000 runs per estimate. To explain such a gain, we first need to introduce the issue of shortcuts computation.

2.3.1 Drawing shortcuts

As stated in [1], the main computational bottleneck for simulating Kleinberg's model comes from the shortcuts.

- There are n^2 shortcuts in the grid (assuming $q = 1$);
- When one wants to a shortcut, any of the $n^2 - 1$ other nodes can be chosen with non-null probability. This can be made by inverse transform sampling, with a cost $\Omega(n^2)$;
- The shortcut distribution depends on the node u considered, even if one uses relative coordinates. For example, a corner node will have $i + 1$ neighbors at distance i for $1 \leq i < n$, against $4i$ neighbors for inner nodes (as long as the ball of radius i stays inside the grid). This means that, up to symmetry, each node has a unique shortcut distribution[1]. This prevents from mutualising shortcuts drawings between nodes.

In the end, building shortcuts as described above for each of the R runs has a time complexity $\Omega(Rn^4)$, which is unacceptable if one wants to evaluate $e_r(n)$ on large grids.

The first issue is easy to address: as observed in [12, 14, 1], we can use the *Principle of deferred decision* [18] and compute the shortcuts on-the-fly as the path is built, because they are drawn independently and a node is never used twice in a given path. This reduces the complexity to $\Omega(Rn^2 e_r(n))$.

2.3.2 Torus approximation

To lower the complexity even more, one can approximate the grid by the torus. This is the approach adopted in [11, 14]. The toroidal topology brings two major features compared to a flat grid:

- The distribution of the relative position of the shortcut does not depend on the originating node. This enables to draw shortcuts in advance (in bulk);
- There is a strong radial symmetry, allowing to draw a "radius" and an "angle" separately.

To illustrate the gain of using a torus instead of a grid, consider the drawing of k shortcuts from k distinct nodes. In a grid, if one uses inverse transform sampling for each corresponding distribution, the cost is $\Omega(n^2 k)$. In the torus, one can compute the probabilities to be at distance i for i between 1 and n (the maximal distance in the torus), draw k radii, then choose for each drawn radius i a node uniformly chosen among those at distance i. Assuming drawing a float uniformly distributed over $[0, 1)$ can be made in $O(1)$, the main bottleneck is the drawing of radii. Using bulk inverse transform sampling, it can be performed in $O(n + k \log(k))$, by sorting k random floats, matching then against the cumulative distribution of radii and reverse sorting the result.

3 Fast Estimation of Expected Delivery Time

We now describe our approach for computing $e_r(n)$ in the flat grid with the same complexity than for the torus approximation.

[1] To take advantage of symmetry, one can consider the isometric group of a square grid, which can be built with the quarter-turn and flip operations. However, its size is 8, so even using symmetry, there are at least $\frac{n^2}{8}$ distinct (non-isomorphic) distributions.

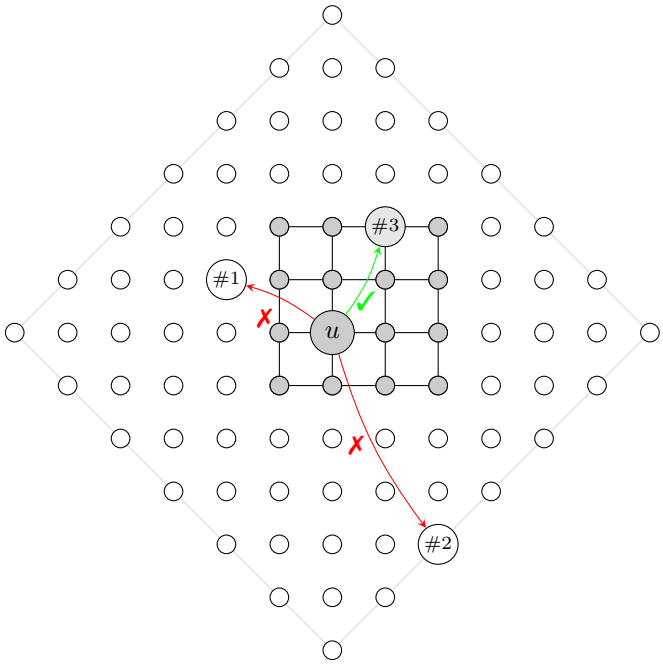


Figure 1 Main idea of the dynamic rejection sampling approach ($n = 4$).

3.1 Dynamic rejection sampling for drawing shortcuts

In order to keep low computational complexity without making any approximation of the model, we propose to draw a shortcut of a node u as follows:

1. We embed the actual grid G (we use here G to refer to the lattice nodes of $G(n, r, p, q)$) in a virtual lattice B_u made of points inside a ball of radius $2(n-1)$. Note that the radius chosen ensures that G is included in B_u no matter the location of u.
2. We draw a node inside B_u such that the probability to pick up a node v is proportional to $(d(u,v))^{-r}$. This can be done in two steps (radius and angle):
   - For the radius, we notice that the probability to draw a node at distance i is proportional to i^{1-r}, so we pick an integer between 1 and $2(n-1)$ such that the probability to draw i is to $i^{1-r} / \sum_{k=1}^{2(n-2)} k^{1-r}$.
   - For the angle, pick an integer uniformly chosen between 1 and $4i$
3. This determines a unique point v among the $4i$ points at distance i from u in the virtual lattice, chosen with a probability proportional to $(d(u,v))^{-r}$. If v belongs to the actual grid, it becomes the shortcut, otherwise we try again (back to step #2).

This technique, illustrated in Figure 1, is inspired by the *rejection sampling* method [21]. By construction, it gives the correct distribution: the node v that it eventually returns is in the actual grid and has been drawn with a probability proportional to $(d(u,v))^{-r}$.

We call this *dynamic* rejection sampling because the sampled distribution changes with the current node u. Considering u as a relative center, the actual grid G moves with u and acts like an acceptance mask. On the other hand, the distribution over the virtual lattice B_u remains constant. This enables to draw batches of relative shortcuts that can be used over multiple runs, exactly like for the torus approximation.

The only possible drawback of this approach is the number of attempts required to draw a correct shortcut. Luckily, this number is contained.

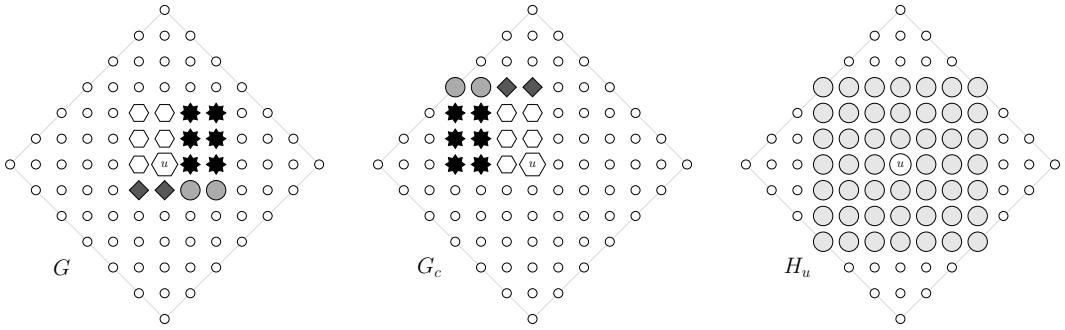


Figure 2 Graphical representation of G, G_c and H_u ($n = 4$). The sub-lattices used to built a bijection between G and G_c (cf. proof of Lemma 1) are represented with distinct shapes and gray levels.

▶ **Lemma 1.** *The probability that a node drawn in B_u belongs to G is at least $\frac{1}{8}$.*

Proof. We will prove that

$$\frac{\sum_{v\in G\setminus\{u\}}(d(u,v))^{-r}}{\sum_{v\in B_u\setminus\{u\}}(d(u,v))^{-r}} > \frac{1}{8}.$$

We use the fact that the probability decreases with the distance combined with some geometric arguments. Let G_c be a $n \times n$ lattice that has u as one of its corner. Let H_u the $(2n-1) \times (2n-1)$ lattice centered in u.

In terms of probability of drawing a node in $G \setminus \{u\}$, the worst case is when u is at some corner: there is a bijection f from G to G_c such that for all $v \in G \setminus \{u\}, d(u, f(v)) \geq d(u, v)$. Such a bijection can be obtained by splitting G into $G \cap G_c$ and three other sub-lattices that are flipped over $G \cap G_c$ (see Figure 2). This gives

$$\sum_{v\in G\setminus\{u\}}(d(u,v))^{-r} \geq \sum_{v\in G\setminus\{u\}}(d(u,f(v)))^{-r} = \sum_{v\in G_c\setminus\{u\}}(d(u,v))^{-r}.$$

Then we observe that the four possible lattices G_c obtained depending on the corner occupied by u fully cover H_u. In fact, axis nodes are covered redundantly. This gives

$$\sum_{v\in H_u\setminus\{u\}}(d(u,v))^{-r} < 4\sum_{v\in G_c\setminus\{u\}}(d(u,v))^{-r}.$$

Lastly, if one folds $B_u \setminus H_u$ back into H_u like the corners of a sheet of paper, we get a strict injection g from $B_u \setminus H_u$ to $H_u \setminus \{u\}$ (the diagonal nodes of H_u are not covered). Moreover, for all $v \in B_u \setminus H_u, d(u, v) \geq d(u, h(v))$. This gives

$$\sum_{v\in B_u\setminus H_u}(d(u,v))^{-r} \leq \sum_{v\in B_u\setminus H_u}(d(u,h(v)))^{-r} < \sum_{v\in H_u\setminus\{u\}}(d(u,v))^{-r}.$$

This concludes the proof, as we get

$$\frac{\sum_{v\in G\setminus\{u\}}(d(u,v))^{-r}}{\sum_{v\in B_u\setminus\{u\}}(d(u,v))^{-r}} \geq \frac{\sum_{v\in G_c\setminus\{u\}}(d(u,v))^{-r}}{\sum_{v\in H_u\setminus\{u\}}(d(u,v))^{-r}+\sum_{v\in B_u\setminus H_u}(d(u,v))^{-r}} > \frac{1}{8}.$$ ◀

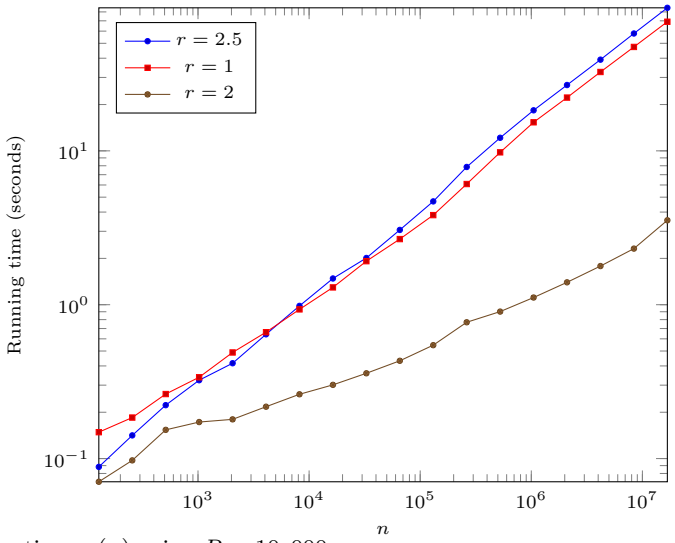


Figure 3 Computing $e_r(n)$ using $R = 10,000$ runs.

Remarks

- When $r = 0$ (uniform shortcut distribution), the bound $\frac{1}{8}$ is asymptotically tight: the success probability is exactly the ratio between the number of nodes in $G \setminus \{u\}$ and $B_u \setminus \{u\}$, which is $\frac{n^2-1}{4(n-1)(2n-1)} \underset{n\to+\infty}{\longrightarrow} \frac{1}{8}$. On the other hand, as r grows, the probability mass gets more and more concentrated around u (hence in G), so we should expect better performance (cf. Section 3.2).
- The dynamic rejection sampling approach can be used in other variants of Kleinberg's model, like for other dimensions or when the number of shortcuts per node is a random variable (like in [9]). The only requirement is the existence of some *root* distribution (like the distribution over B_u here) that can be carved to match any of the possible distributions with a simple acceptance test.
- Only the nodes from H_u may belong to G, so nodes from $B_u \setminus G$ are always sampled for nothing. For example, in Figure 1, this represents 36 nodes over 84. By drawing shortcuts in $H_u \setminus \{u\}$ instead of $B_u \setminus \{u\}$, we could increase the success rate lower bound to 1/4. However, this would make the algorithm implementation more complex (the number of nodes at distance i is not always $4i$), which is in practice not worth the factor 2 improvement of the lower bound. This may not be true for a higher dimension β. Adapting the proof from Lemma 1, we observe that using a ball of radius $\beta(n-1)$ will lead to a bound $\beta!(2\beta)^{-\beta}$, while a grid of side $2n-1$ will lead to $2^{-\beta}$. The two bounds are asymptotically tight for $r = 0$. In that case the grid approach is $\frac{\beta^\beta}{\beta!}$ more efficient than the ball approach (this grows exponentially with β).

3.2 Performance Evaluation

We implemented our algorithm in Julia 0.4.5. A working code example is provided in Appendix A. Simulations were executed on a low end device (a Dell tablet with 4 Gb RAM and Intel M-5Y10 processor), which was largely sufficient for getting fast and accurate results on very large grids, thanks to the dynamic rejection sampling.

Unless said otherwise, $e_r(n)$ is obtained by averaging $R = 10,000$ runs. For n, we mainly consider powers of two ranging from 2^7 (about 16,000 nodes) to 2^{24} (about 280 trillions nodes). We focus on $r \in [0, 3]$.

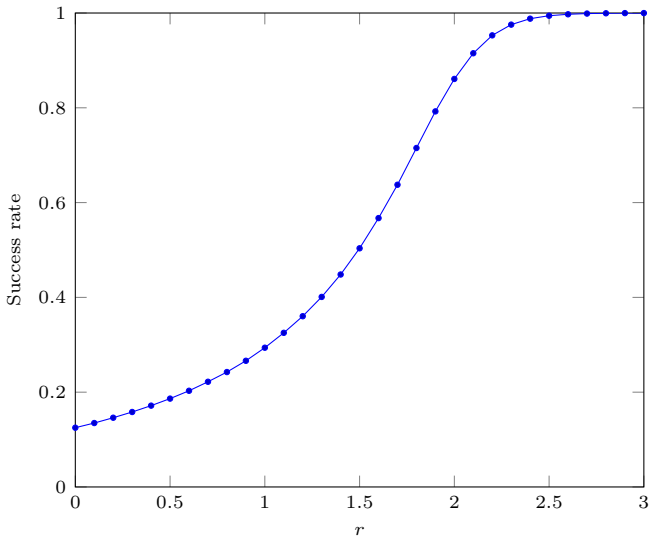


Figure 4 Frequency of shortcuts drawn in B_u that belongs to G observed during a computation of $e_r(n)$ ($n = 2^{14}$).

Regarding the choice of the bulk size k (we assume here the use of bulk inverse transform sampling), the average number of shortcuts to draw over the R runs is $Re_r(n)$. This leads to an average total cost for drawing shortcuts in $O(\lceil \frac{Re_r(n)}{k} \rceil (n + k\log(n)))$. k can be optimized if $e_r(n)$ is known, but this requires bootstrapping the estimation of $e_r(n)$. Yet, we can remark that for n fixed and R large enough, we should choose k of the same order of magnitude than n, which ensures an average cost per shortcut in $O(\log(n))$. This is the choice we made in our code, which gives a complexity in $O(\max(Re_r(n), n)\log(n))$. This is not efficient for $n \gg Re_r(n)$ (the bulk is then over-sized, so lot of unused shortcuts are drawn), but this seldom happens with our settings.

Figure 3 presents the time needed to compute $e_r(n)$ as a function of n for $r \in \{1, 2, \frac{5}{2}\}$. We observe running times ranging from seconds to a few minutes. To compare with, in [1], which is to the best of our knowledge the only work disclosing computation times, one single run takes about 4 seconds for $n = 400$. With a similar time budget, our implementation averages 10,000 runs for $n = 2^{17} \approx 130,000$ ($r = 1$ or $r = 2.5$), or up to $n = 2^{24} \approx 16,000,000$ for the optimal exponent $r = 2$. In other words, we are several orders of magnitude faster.

We also looked at the cost of the dynamic rejection sampling approach in terms of shortcuts drawn outside of G. Figure 4 shows the success frequency of the sampling as a function of r for $n = 2^{14}$ (the actual value of n has no significant impact as long as it is large enough). We verify Lemma 1: the success rate is always at least $1/8 = 0.125$, which is a tight bound for $r = 0$. For $r = 1$, the rate is about 0.29, and it climbs to 0.86 for $r = 2$. Failed shortcuts become negligible (rate greater than 0.99) for $r \geq 2.5$. Overall, Figure 4 shows that the cost of drawing some shortcuts outside G is a small compared to the benefits offered by drawing shortcuts from a unique distribution.

▶ Remark. In terms of success rate, Figure 4 shows that $r = 2.5$ is much more efficient than $r = 1$, but running times tend to be slightly longer for $r = 2.5$ (cf. Figure 3). The reason is that $e_1(n)$ is lower than $e_{2.5}(n)$, which overcompensates the success rate difference.

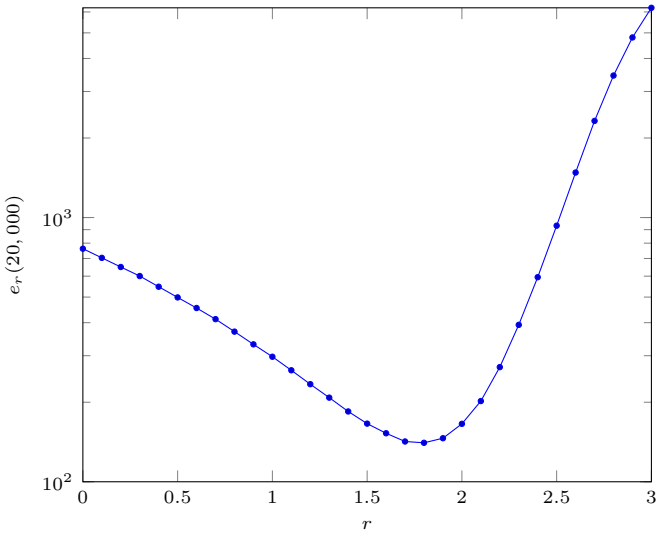


Figure 5 Expected delivery time for $n = 20,000$.

4 Applications

Given the tremendous amount of strong theoretical results on small-world routing, one can question the interest of proposing a simulator (even a fast one!).

In this Section, we prove the interest of numerical evaluation through three (almost) independent small studies.

4.1 Efficient enough exponents

In [11], Kleinberg gives an estimation of $e_r(20,000)$ using a torus approximation (cf. Section 2.3.2). A few years later, he discussed in more details the results, observing that [5]:

- $e_r(n)$ stays quite similar for $r \in [1.5, 2]$;
- the best value of r is actually slightly lower than 2.

We believe that these observations are very important as they show that routing in Kleinberg's grid is more robust than predicted by theory: it is efficient as long as r is *close enough* to 2.

Using our simulator, we can perform the same experiment. As shown by Figure 5, the results are quite similar to the ones observed in [11].

Yet, there is a small but *essential* difference between the two experiments: Kleinberg approximated his grid by a torus while we stay true to the original model. Why do we use the word *essential*? Both shapes have the same asymptotic behavior (proofs in [11] are straightforward to adapt), so why should we care? It seems to us that practical robustness is an essential feature if one wants to accept Kleinberg's grid as a reasonable model for routing in social networks. To the best of our knowledge, no theoretical work gives quantitative results on this robustness, so we need to rely on numerical evaluation. But when Kleinberg uses a torus approximation, we can not rule out that the observed robustness is a by-side effect of the toroidal topology. The observation of a similar phenomenon on a flat grid discards this hypothesis. In fact, it suggests (without proving) that the robustness with respect to the exponent for grids of finite size may be a general phenomenon.

We propose now to investigate this robustness in deeper details. We have evaluated the following values, which outline, for a given n, the values of r that can be considered

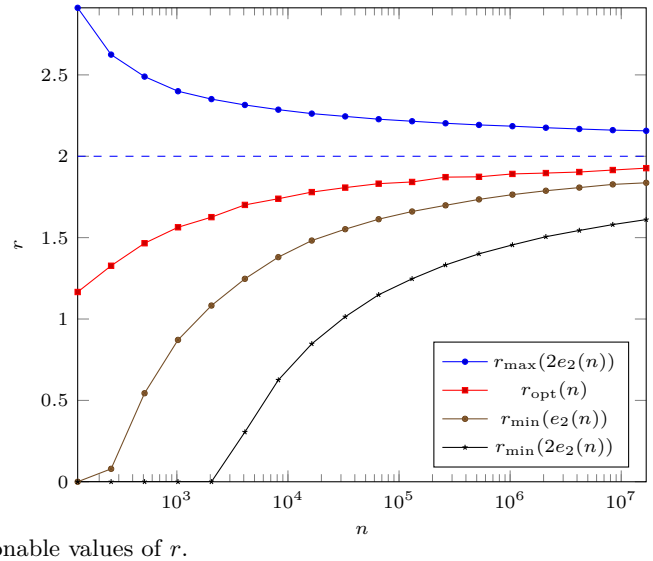


Figure 6 Reasonable values of r.

reasonable for performing greedy routing:

- The value of r that minimizes $e_r(n)$, denoted r_{opt};
- The smallest value of r such that $e_r(n) \leq e_2(n)$, denoted $r_{\min}(e_2(n))$;
- The smallest and largest values of r such that $e_r(n) \leq 2e_2(n)$, denoted $r_{\min}(2e_2(n))$ and $r_{\max}(2e_2(n))$ respectively.

The results are displayed in Figure 6. All values but r_{opt} are computed by bisection. For r_{opt}, we use a Golden section search [19]. Finding a minimum requires more accuracy, so the search of r_{opt} is set to use $R = 1,000,000$ runs per estimation. Luckily, as the computation operates by design through near-optimal values, we can increase the accuracy with reasonable running times.

Besides confirming that $r = 2$ is asymptotically the optimal value, Figure 6 shows that the range of reasonable values for finite grids is quite comfortable. For example, considering the range of values where $e_r(n)$ is less than twice $e_2(n)$, we observe that:

- For $n \leq 2^{11}$ (less than four million nodes), any r between 0 and 2.35 works;
- For $n \leq 2^{14}$ (less than 270 million nodes), the range is between 0.85 and 2.26.
- Even for $n = 2^{24}$ (about 280 trillions nodes), all values of r between 1.58 and 2.16 can still be considered *efficient enough*.

4.2 Asymptotic Behavior

Our simulator can be used to verify the theoretical bounds proposed in [12]. For example, Figure 7 shows $e_r(n)$ for r equal to 1, 2, and $\frac{5}{2}$.

As predicted, $e_r(n)$ seems to behave like $\log^2(n)$ for $r = 2$ and like n^α for the two other cases. Yet, the exponents found differ from the ones proposed in [12]. For both $r = 1$ and $r = 2.5$, we observe $\alpha = \frac{1}{2}$, while the lower bound is $\frac{1}{3}$. Intrigued by the difference, we want to compute α as a function of r.

However, we see in Figure 7 that a $\log^2(n)$ curve appears to have a positive slope in a logarithmic scale, even for large values of n. This may distort our estimations. To control the possible impact of this distortion, we estimate the exponent at two distinct scales:

- $n \in [2^{15}, 2^{20}]$, using the estimation $\alpha_1 := \frac{\log_2(e_r(2^{20})) - \log_2(e_r(2^{15}))}{5}$;
- $n \in [2^{20}, 2^{24}]$, using the estimation $\alpha_2 := \frac{\log_2(e_r(2^{24})) - \log_2(e_r(2^{20}))}{4}$.

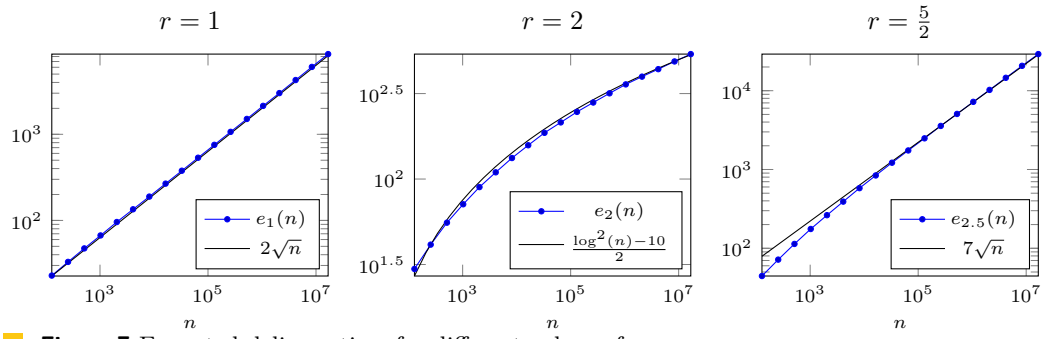


Figure 7 Expected delivery time for different values of r.

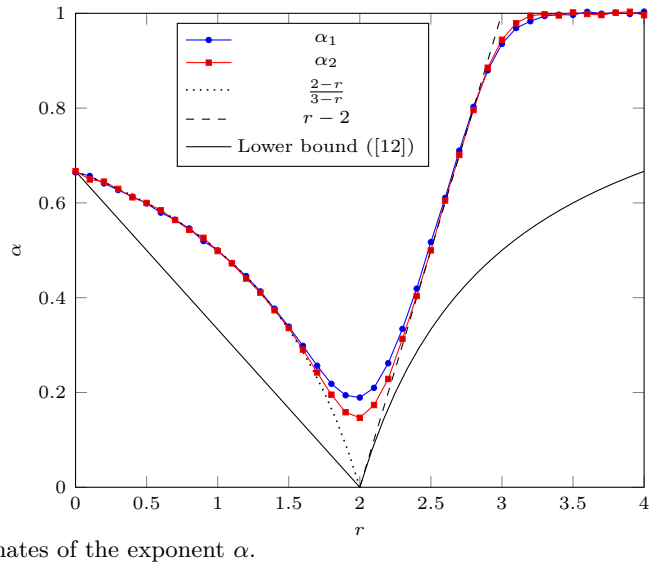


Figure 8 Estimates of the exponent α.

The results are displayed in Figure 8. The range of r was extended to $[0, 4]$ due to the observation of a new transition for $r = 3$

As feared, our estimations do not indicate 0 for $r = 2$, but the fact that α_2 is closer to 0 than α_1 confirms that this is likely caused by the $\log^2(n)$ factor. Moreover, we observe that α_1 and α_2 only differ for $r \approx 2$ and $r \approx 3$, suggesting that both estimates should be accurate except for these critical values.

Based on Figures 7 and 8, it seems that the bounds proposed by Kleinberg in [12] are only tight for $r = 0$ and $r = 2$. Formally proving more accurate bounds is beyond the scope and spirit of this paper, but we can conjecture new bounds hinted by our simulations:

- For $0 \leq r < 2$, we have $e_r(n) = \Theta(n^{\frac{2-r}{3-r}})$;
- For $2 < r < 3$, we have $e_r(n) = \Theta(n^{r-2})$;
- For $r > 3$, we have $e_r(n) = \Theta(n)$.

This conjectured bounds are consistent with the one proposed in [2], where it is proved that for the 1-dimensional ring, we have:

- For $0 \leq r < 1$, $e_r(n) = \Omega(n^{\frac{1-r}{2-r}})$;
- For $r = 1$, $e_r(n) = \Theta(\log^2(n))$
- For $1 < r < 2$, $e_r(n) = O(n^{r-1})$;
- For $r = 2$, $e_r(n) = O(n^{\frac{\log(\log(n))}{\log(n)}})$
- For $r > 2$, $e_r(n) = O(n)$.

It is likely that the proofs in [2] can be adapted to the 2-dimensional grid, but some additional work may be necessary to demonstrate the bounds' tightness. Moreover, our estimates may have missed some slower-than-polynomial variations. For example, the logarithms in the bounds for $r = 2$ in [2] may have a counterpart in the grid for $r = 3$. This would explain why the estimates are not sharp around that critical value (like for the case $r = 2$ and the term in $\log^2(n)$).

▶ Remark. For $r \geq 3.5$, we have observed that $e_r(n) \approx \frac{2}{3}n$, which is the expected delivery path in absence of shortcuts: for these high values of r, almost all shortcuts link to an immediate neighbor of the current node, which make them useless, and the rare exceptions are so short that they make no noticeable difference.

4.3 Six degrees of separation

What makes Milgram's experiments [17, 20] so famous is the surprising observation that only a few hops are required to transmit messages in social networks, a phenomenon called *six degrees of separation* in popular culture.

Yet, the values of $e_r(n)$ observed until now are quite far for the magic number six. For example, in Figure 5, the lowest value is 140. In addition to the asymptotic lack of robustness with respect to the exponent (discussed in Section 4.1), this may partially motivate the amount of work accomplished to increase the realism of the model and the performance of the routing algorithm.

In our opinion, a good model should be as simple as possible while being able to accurately predict the phenomenon that needs to be explained. We propose here to answer a simple question: is the genuine model of greedy routing in $G(n, r, p, q)$ a good model? Sure, it is quite simple. To discuss its accuracy, we need to tune the four parameters n, r, p and q to fit the conditions of Milgram's experiments as honestly as we can.

Size. The experiments of Milgram were conducted in the United States, with a population of about 200,000,000 at the late sixties. All inhabitants were not susceptible to participate to the experiments: under-aged, undergraduate or disadvantaged people may be considered as *de facto* excluded from the experiments. Taking that into consideration, the correct n is probably somewhere in the range $[5000, 14000]$. We propose to set $n = 8,500$, which corresponds to about 72,000,000 potential subjects.

Exponent. In [5], Kleinberg investigates how to relate the r-harmonic distribution with real-life observations. He surveys multiple social experiments and discusses the correspondence with the exponent of his model, which gives estimates of r between 1.75 and 2.2.

Neighborhood. The default value $p = q = 1$ means that there are no more than five "acquaintances" per node. This is quite small compared to what is observed in real-life social networks. For example, the famous Dunbar's number, which estimates the number of *active* relationships, is 150 [4]. More recent studies seem to indicate that the average number of acquaintances is larger, ranging from 250 to 1500 (see [3, 16, 22] and references within). We propose to set p and q so that the neighborhood size $2p(p + 1) + q$ is about 600, the value reported in [16]. Regarding the partition between local links (p) and shortcuts (q), we consider three typical scenarios:

- $p = 1$, $q = 600$ (shortcut scenario: the neighborhood is almost exclusively made of shortcuts, and local links are only here to ensure the termination of greedy routing).
- $p = 10$, $q = 380$ (balanced scenario).
- $p = 15$, $q = 120$ (local scenario, with a value of q not too far from Dunbar's number).

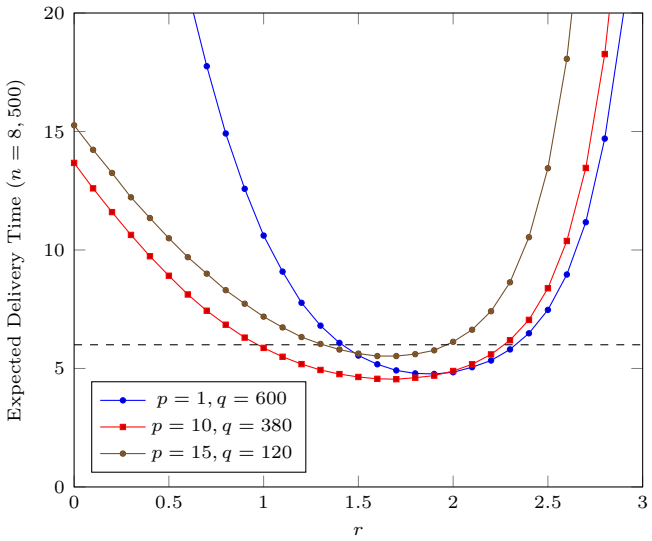


Figure 9 Performance of greedy routing for parameters inspired by Milgram's experiments.

Having set all parameters, we can evaluate the performance of greedy routing. The results are displayed in Figure 9. We observe that the expected delivery time roughly stands between five and six for a wide range of exponents.

- $r \in [1.4, 2.3]$ for the shortcut scenario.
- $r \in [1.3, 2.3]$ for the balanced scenario.
- $r \in [1.3, 2]$ for the local scenario.

Except for the local scenario, which leads to slightly higher routing times for $r > 2$, the six degrees of separation are achieved for all values of r that are consistent with the observations surveyed in [5]. This allows to answer our question: the augmented grid proposed by Kleinberg is indeed a good model to explain the *six degrees of separation* phenomenon.

5 Conclusion

We proposed an algorithm to evaluate the performance of greedy routing in Kleinberg's grid. Fueled by a dynamic rejection sampling approach, the simulator based on our solution performs several orders of magnitude faster than previous attempts. It allowed us to investigate greedy routing under multiple perspective.

- We noted that the performance of greedy routing is less sensitive to the choice of the exponent r than predicted by the asymptotic behavior, even for very large grids.
- We observed that the bounds proposed in [12] are not tight except for $r = 0$ and $r = 2$. We conjectured that the tight bounds are the 2-dimensional equivalent of bounds proposed in [2] for the 1-dimensional ring.
- We claimed that the model proposed by Kleinberg in [12, 11] is a good model for the *six degrees of separation*, in the sense that it is very simple *and* accurate.

Our simulator is intended as a tool to suggest and evaluate theoretical results, and possibly to build another bridge between theoretical and empirical study of social systems. We hope that it will be useful for researchers from both communities. In a future work, we plan to make our simulator more generic so it can handle other types of graph and augmenting schemes.

References

1 Stavros Athanassopoulos, Christos Kaklamanis, Ilias Laftsidis, and Evi Papaioannou. An experimental study of greedy routing algorithms. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 150–156, June 2010.

2 Lali Barrière, Pierre Fraigniaud, Evangelos Kranakis, and Danny Krizanc. Efficient routing in networks with long range contacts. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC'01, pages 270–284, London, UK, 2001.

3 Ithiel de Sola Pool and Manfred Kochen. Contacts and influence. *Social Networks*, 1:5–51, 1978.

4 R. I. M. Dunbar. Neocortex size as a constraint on group size in primates. *Journal of Human Evolution*, 22(6):469–493, June 1992.

5 David Easley and Jon Kleinberg. The small-world phenomenon. In *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*, chapter 20, pages 611–644. Cambridge University Press, 2010.

6 Pierre Fraigniaud, Cyril Gavoille, Adrian Kosowski, Emmanuelle Lebhar, and Zvi Lotker. Universal Augmentation Schemes for Network Navigability: Overcoming the $\sqrt{(n)}$-Barrier. *Theoretical Computer Science*, 410(21-23):1970–1981, 2009.

7 Pierre Fraigniaud, Cyril Gavoille, and Christophe Paul. Eclecticism shrinks even small worlds. *Distributed Computing*, 18(4):279–291, 2006.

8 Pierre Fraigniaud and George Giakkoupis. On the searchability of small-world networks with arbitrary underlying structure. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 389–398, June 6–8 2010.

9 Pierre Fraigniaud and George Giakkoupis. Greedy routing in small-world networks with power-law degrees. *Distributed Computing*, 27(4):231–253, 2014.

10 Frigyes Karinthy. Láncszemek, 1929.

11 Jon Kleinberg. Navigation in a small world. *Nature*, August 2000.

12 Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *in Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 163–170, 2000.

13 David Liben-Nowell, Jasmine Novak, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Geographic routing in social networks. *Proceedings of the National Academy of Sciences of the United States of America*, 102(33):11623–11628, 2005.

14 Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy neighbor's neighbor: The power of lookahead in randomized P2P networks. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing (STOC)*, pages 54–63. ACM, 2004.

15 Chip Martel and Van Nguyen. Analyzing kleinberg's (and other) small-world models. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC'04, pages 179–188, New York, NY, USA, 2004. ACM.

16 Tyler H. McCormick, Matthew J. Salganik, and Tian Zheng. How many people do you know?: Efficiently estimating personal network size. *Journal of the American Statistical Association*, 105(489):59–70, 2010.

17 Stanley Milgram. The small world problem. *Psychology Today*, 67(1):61–67, 1967.

18 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

19 William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Minimization or maximization of functions. In *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, chapter 10. Cambridge University Press, 3 edition, 2007.

20 Jeffrey Travers and Stanley Milgram. An experimental study of the small world problem. *Sociometry*, 32:425–443, 1969.

21 John von Neumann. Various techniques used in connection with random digits. *Nat. Bureau Standards*, 12:36–38, 1951.

22 Barry Wellman. Is Dunbar's number up? *British Journal of Psychology*, 2011.

A Code for Expected Delivery Time (tested on Julia Version 0.4.5)

```
using StatsBase # For using Julia built-in sample function

function shortcuts_bulk(n, probas, bulk_size)
  radii = sample(1:(2*n-2), probas, bulk_size)
  shortcuts = Tuple{Int, Int}[]
  for i = 1:bulk_size
    radius = radii[i]
    angle = floor(4*radius*rand())-2*radius
    push!(shortcuts, ((radius-abs(angle)),
      (sign(angle)*(radius-abs(radius-abs(angle))))))
  end
  return shortcuts
end

# Estimates the Expected Delivery Time for G(n, r, p, q) over R runs
function edt(n, r, p, q, R)
  bulk_size = n
  probas = weights(1./(1:(2*n-2)).^(r-1))
  shortcuts = shortcuts_bulk(n, probas, bulk_size)
  steps = 0
  for i = 1:R
    # s: start/current node; a: target node; d: distance to target
    s_x, s_y, a_x, a_y = tuple(rand(0:(n-1), 4)...)
    d = abs(s_x - a_x) + abs(s_y - a_y)
    while d>0
      sh_x, sh_y = -1, -1 # sh will be best shortcut node
      d_s = 2*n # d_s will be distance from sh to a
      for j = 1:q # Draw q shortcuts
        ch_x, ch_y = -1, -1 # ch will be current shortcut
        c_s = 2*n # c_s will be distance from ch to a
        # Dynamic rejection sampling
        while (ch_x < 0 || ch_x >= n || ch_y < 0 || ch_y >= n)
          r_x,r_y = pop!(shortcuts)
          ch_x, ch_y = s_x + r_x,  s_y + r_y
          if isempty(shortcuts)
            shortcuts = shortcuts_bulk(n, probas, bulk_size)
          end
        end
        c_s = abs(a_x - ch_x) + abs(a_y - ch_y)
        if c_s < d_s # maintain best shortcut found
          d_s = c_s
          sh_x, sh_y = ch_x, ch_y
        end
      end
      if d_s < d-p # Follow shortcut if efficient
        s_x, s_y  = sh_x, sh_y
        d = d_s
      else # Follow local links
        d = d - p
        delta_x = min(p,abs(a_x - s_x))
        delta_y = p - delta_x
        s_x += delta_x*sign(a_x - s_x)
        s_y += delta_y*sign(a_y - s_y)
      end
      steps += 1
    end
  end
  steps /= R
  return steps
end
```

Generalized Selectors and Locally Thin Families with Applications to Conflict Resolution in Multiple Access Channels Supporting Simultaneous Successful Transmissions

Annalisa De Bonis

Università di Salerno, Fisciano, Italy
debonis@dia.unisa.it

Abstract

We consider the *Conflict Resolution Problem* in the context of a multiple-access system in which several stations can transmit their messages simultaneously to the channel. We assume that there are n stations and that at most k, $k \leq n$, stations are *active* at the same time, i.e, are willing to transmit a message over the channel. If in a certain instant at most d, $d \leq k$, active stations transmit to the channel then their messages are successfully transmitted, whereas if more than d active stations transmit simultaneously then their messages are lost. In this latter case we say that a *conflict* occurs. The present paper investigates *non-adaptive* conflict resolution algorithms working under the assumption that active stations receive a *feedback* from the channel that informs them on whether their messages have been successfully transmitted. If a station becomes aware that its message has been correctly sent over the channel then it becomes immediately *inactive*, that is, stops transmitting. The measure to optimize is the number of time slots needed to solve conflicts among all active stations. The fundamental question is how much this measure decreases with the number d of messages that can be simultaneously transmitted with success. In this paper we prove that it is possible to achieve a speedup linear in d by providing a conflict resolution algorithm that uses a $1/d$ ratio of the number of time slots used by the optimal conflict resolution algorithm for the particular case $d = 1$ [20]. Moreover, we derive a lower bound on the number of time slots needed to solve conflicts non-adaptively which is within a $\log(k/d)$ factor from the upper bound. To the aim of proving these results, we introduce a new combinatorial structure that consists in a generalization of Komlós and Greenberg codes [20]. Constructions of these new codes are obtained via a new kind of selectors [11], whereas the non-existential result is implied by a non-existential result for a new generalization of the locally thin families of [1, 10]. We believe that the combinatorial structures introduced in this paper and the related results may be of independent interest.

1998 ACM Subject Classification C.2.2 Network Protocols

Keywords and phrases Multiple-Access channels, Multi Access Communication, Conflict Resolutions, New Combinatorial Structures, Selectors

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.22

1 Introduction

Conflict resolution is a fundamental problem in multiple-access communication and has been widely investigated in the literature both for its practical implications and for the many theoretical challenges it poses [6]. Commonly, this problem is studied under the assumption of the so called *collision model* in which simultaneous transmission attempts by two or more stations result in the destruction of all messages. However, as already observed in [16] and

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 22; pp. 22:1–22:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

more recently in [5], this restrictive multiaccess model does not capture the features of many important multiuser communication systems in which several messages can be successfully transmitted at the same time. Examples of communication systems allowing multiple-packet reception include Code Division Multiple Access (CDMA) systems in which several stations share the same frequency band, and Multiple-Input Multiple-Output (MIMO) systems, that enhance the capacity of a radio link by using multiple antennas at the transmitter and the receiver. These systems are largely used in the phone standards, satellite communication systems, and in wireless communication networks. Multiple-packet reception is also achieved through coding techniques specifically designed for copying with collisions. Recently, the authors of [4] introduced a coding technique, for the finite-field additive radio network model, that enables broadcast in a network with a bounded number of transmitters. These codes have the property that, when codewords from at most a certain number of different transmitting nodes are summed up, then the receiving nodes are able to recover the original transmissions.

Given the growing relevance of systems allowing multiple-packet reception in modern communication technologies, it is crucial to consider multiple-access models that better capture the phenomenon occurring when several packets can be sent simultaneously over the channel. The following quotation from [5] well emphasizes the importance of these communication models: "Traditionally, practical design and theoretical analysis of random multiple access protocols have assumed the classical collision channel model – namely, a transmitted packet is considered successfully received as long as it does not overlap or 'collide' with another. Although this model is analytically amenable and reflected the state of technology when networking was an emerging field, the classical collision model does not represent the capabilities of today's transceivers. In particular, present transceiver technologies enable users to correctly receive multiple simultaneously transmitted data packets. With proper design, this capability – commonly referred to as multiple packet reception (MPR) [17, 16] – can significantly enhance network performance.".

Communication models allowing multiple simultaneous successful transmissions have received great attention in the literature in recent times [5, 12, 14, 18, 22, 24, 25]. The following fundamental question arises when studying conflict resolution in the above described models: How fast does the number of time slots needed to solve conflicts decrease with the number d of messages that can be simultaneously transmitted with success? In this paper we show that it is possible to achieve a speedup linear in d when dealing with multiple-access systems with feedback, i.e., systems in which whenever an active station transmits to the channel, it receives a feedback that informs the station on whether its transmission has been successful.

1.1 The Model and Related Work

We consider a multiple-access system in which n stations have access to the channel and at most $k \leq n$ stations are willing to transmit a message at the same time. We call these stations *active* stations. If at most $d \leq k$ active stations transmit to the channel then these stations succeed to transmit their messages, whereas if more than d stations transmit then all messages are lost. In this latter case, we say that a *conflict* occurs. We assume that time is divided into time slots and that transmissions occur during these time slots. We also assume that all stations have a global clock and that active stations start transmitting at the same time slot. A scheduling algorithm for such a multiaccess system is a protocol that schedules the transmissions of the n stations over a certain number t of time slots (*steps*) identified by integers $1, 2, \ldots, t$. Whenever an active station transmits to the channel, it

receives a feedback from the channel that informs the station on whether its transmission has been successful. As soon as an active station becomes aware that its message has been successfully transmitted, it becomes *inactive* and does not transmit in the following time slots, even though it is scheduled to transmit by the protocol. For the particular case $d = 1$, our model corresponds to the multiple-access model with feedback considered by Komlós and Greenberg in [20].

In this paper we focus on non-adaptive scheduling algorithms, that is, algorithms that schedule all transmissions in advance so that all stations transmit according to a predetermined protocol known to them from the very beginning. Please notice that the knowledge of the feedback cannot affect the schedule of transmissions but can only signal a station to become inactive after it has successfully transmitted. A non-adaptive scheduling algorithm is represented by a $t \times n$ Boolean matrix where each column is associated with a distinct station and a station j is scheduled to transmit at step i if and only if entry (i, j) of the matrix is 1. In fact station j really transmits at step i if and only it is an active station and is scheduled to transmit at that step.

A *conflict resolution algorithm* is a scheduling protocol that schedules transmissions in such a way that all active stations transmit with success, i.e., for each active station there is a time slot in which it is scheduled to transmit on the channel and at most $d - 1$ other active stations are allowed to transmit in that time slot. The conflict resolution protocols considered in this paper are non-adaptive. The parameter we are interested in minimizing is the number of rows of the matrix which corresponds to the number of time slots over which the conflict resolution algorithm schedules the transmissions.

For the particular case $d = 1$, Komlós and Greenberg [20] gave a non-adaptive protocol that uses $O\left(k \log \frac{n}{k}\right)$ time slots to solve all conflicts among up to k active stations. Later on, the authors of [11, 21] proved the same upper bound by providing a simple construction based on selectors [11]. The above upper bound has been shown to be the best possible in [9], and later on, independently by the authors of [8, 10]. The lower bound in [8, 9, 10] improved on the $\Omega\left(\frac{k}{\log k} \log n\right)$ lower bound in [19], which additionally holds for adaptive algorithms that however are not the topic of this paper.

In [12] it has been studied the *no-feedback* version of the multiple-access problem considered in the present paper, i.e., the scenario in which at most d out of up to k active stations can transmit their messages simultaneously with success and the stations receive no feedback from the channel. That paper provides both upper and lower bounds on the minimum number of time slots needed to solve conflicts in the no-feedback model. The lower bound has been later improved by a combinatorial result given in [13]. Interestingly, the upper bound of [12] and the lower bound of [13] exceed, respectively, our upper and lower bounds for multiple-access systems with feedback by a $\frac{k}{d}$ factor.

1.2 Our results

In this paper we investigate the conflict resolution problem under the multiaccess model described in the previous section. To this aim, we introduce a new generalization of Komlós and Greenberg codes [20]. We prove that these new codes are equivalent to scheduling algorithms that allow up to k active stations to transmit with success in our setting, thus showing that upper and lower bounds on the minimum length of these codes translate into upper and lower bounds on the minimum number of time slots needed to solve conflicts. We present upper and lower bounds of the minimum length of these codes that differ asymptotically by a $\log(k/d)$ factor. These bounds are a consequence of the corresponding

bounds for other two new combinatorial structures also introduced in this paper. In particular, the proposed construction of generalized Komlós and Greenberg codes is based on a new version of (k, m, n)-selectors [11] having an additional parameter d. We give an existential result for this version of selectors based on the Lovász Local Lemma and present a Moser-Tardos type randomized algorithm to generate selectors meeting the proved upper bound. The lower bound follows from a non-existential result for a new combinatorial structure that can be regarded as an extension of the selective families of [3, 7] and the $\leq k$-locally thin codes of [10]. We call these new structures $(\leq k, d, n)$-locally thin codes.

Our main results are summarized by the following theorems.

▶ **Theorem 1.** *Let k, d, and n be integers such that $1 \leq d \leq k \leq n$. There exists a conflict resolution algorithm for a multiple-access channel with feedback that schedules the transmissions of n stations in such a way that all active stations transmit with success, provided that the number of active stations is at most k and that the channel allows up to d stations to transmit their messages simultaneously with success. The number of time slots t used by this algorithm is*

$$t = O\left(\frac{k}{d}\log\frac{n}{k}\right).$$

▶ **Theorem 2.** *Let k, d, and n be positive integers such that $3(d+1) \leq k \leq n$. Let $\mathcal{A}$ be* any *conflict resolution algorithm for a multiple-access channel with feedback that schedules the transmissions of n stations in such a way that all active stations transmit with success, provided that the number of active stations is at most k and that the channel allows up to d stations to transmit their messages simultaneously with success. The number of time slots t needed by $\mathcal{A}$ is*

$$t = \Omega\left(\frac{k}{d\log(k/d)}\log\frac{n}{k(d+1)}\right).$$

We remark that the asymptotic upper bound of Theorem 1 holds also in the case when there is no a priori knowledge of the number k of active stations. In this case, conflicts are resolved by running the conflict resolution algorithm of Theorem 1 iteratively (in stages), each time doubling the number of stations that are assumed to be active. In other words, at stage i the conflict resolution algorithm of Theorem 1 is run for a number k_i of supposedly active stations equal to 2^i. At stage $\lceil \log k \rceil$, the algorithm of Theorem 1 is run for a number of active stations larger than or equal to k and we are guaranteed that all active stations transmit with success within that stage.

Our paper is organized as follows. In Section 2 we introduce the fundamental combinatorial tools. We first introduce the new generalization of Komlós and Greenberg codes and prove that these new codes are equivalent to conflict resolution algorithms for our problem. Then, we introduce our generalized version of selectors and describe how to obtain a conflict resolution protocol by exploiting these combinatorial structures. We conclude Section 2 by giving the definition of $(\leq k, d, n)$-locally thin codes and show that our generalized version of Komlós and Greenberg codes is indeed a $(\leq k, d, n)$-locally thin code, thus proving that any non-existential result for $(\leq k, d, n)$-locally thin codes implies a non existential result for the conflict resolution protocols in our model. In Section 3 we provide existential results for generalized selectors and exploit these existential results to obtain the upper bound of Theorem 1. In Section 3 we also give a randomized algorithm to generate selectors meeting the proved upper bound. In Section 4 we give a lower bound on the minimum length of

$(\leq k, d, n)$-locally thin codes that implies the lower bound of Theorem 2. In that section, we also present a non-existential result for a combinatorial structure satisfying a weaker property than that of $(\leq k, d, n)$-locally thin codes and that can be regarded as a generalization of the k-locally thin families of [1]. Besides its combinatorial interest, this result implies a lower bound on the number of times slots needed to solve conflicts when the number of active stations is known to be *exactly* equal to k.

Due to space limit, some of the proofs are omitted in the present version of the paper.

2 Combinatorial Structures

In the following, for a positive integer m, we denote by $[m]$ the set $\{1, 2, \ldots, m\}$. Given a matrix M, we denote the set of its columns and the set of its column indices by M itself. The rows of a $t \times n$ matrix are numbered from the top to the bottom with integers from 1 to t. The n stations are identified by integers from 1 through n and for a given subset $S \subseteq [n]$ and an n-column matrix M, we denote by $M[S]$ the submatrix formed by the columns of M with indices in S.

2.1 Generalized Komlós and Greenberg Codes

▶ **Definition 3.** Let k, d, and n be integers such that $1 \leq d \leq k \leq n$. A $t \times n$ Boolean matrix is said to be a KG (k, d, n)-code of length t if for any submatrix M' of k columns of M there exists a non-empty set of row indices $\{i_1, \ldots, i_\ell\} \subseteq [t]$, with $i_1 < i_2 < \ldots < i_\ell$, such that the following property holds.

> There exists a partition $\{M'_1, \ldots, M'_\ell\}$ of the set of columns of M' such that, for $j = 1, \ldots, \ell$, $1 \leq |M'_j| \leq d$ and the i_j-th row of M' has all entries at the intersection with the columns of M'_j equal to 1 and those at the intersection with the columns in $M'_{j+1}, \ldots, M'_\ell$ equal to 0.

We will denote by $t_{KG}(k, d, n)$ the minimum length of a KG (k, d, n)-code.

The following theorem states that a KG (k, d, n)-code is indeed equivalent to a scheduling protocol for our multiple-access model with feedback that allows all up to k active stations to transmit with success.

▶ **Theorem 4.** *Let k, d, and n be integers such that $1 \leq d \leq k \leq n$, and let $\mathcal{A}$ be a scheduling algorithm for a multiple-access channel with feedback that allows up to d stations to transmit their messages simultaneously with success. $\mathcal{A}$ is a conflict resolution algorithm that schedules the transmissions of n stations in such a way that all of the up to k active stations transmit with success, if and only if the Boolean matrix corresponding to $\mathcal{A}$ is a KG (k, d, n)-code.*

Proof. The proof is omitted and will be given in the extended version of the paper. ◀

In the following two sections we introduce our generalized versions of selectors and locally thin codes and unveil their relationships with KG (k, d, n)-codes and, consequently, with our conflict resolution problem.

2.2 Generalized Selectors

The following definition introduces a new combinatorial structure that will be employed as a building block to construct KG (k, d, n)-codes. This new structure generalizes the notion of (k, m, n)-selectors introduced in [11] and corresponds to this notion for $d = 1$.

▶ **Definition 5.** Let k, m, d, and n be integers such that $1 \leq d \leq m \leq k \leq n$. A $t \times n$ Boolean matrix is said to be a (k, m, d, n)-selector if any k-column submatrix M' of M contains a set R of rows such that each row in R has Hamming weight comprised between 1 and d, and the Boolean sum of all rows of R has Hamming weight at least m. The number of rows t of the (k, m, d, n)-selector is the size of the selector. The minimum size of (k, m, d, n)-selectors is denoted by $t_{sel}(k, m, d, n)$.

A (k, m, d, n)-selector defines a scheduling algorithm for our multiaccess model that, in the presence of up to k active stations, allows all but at most $k - m$ of these stations to transmit with success. Indeed, all active stations that are scheduled to transmit in the time slots corresponding to the rows in R, transmit with success, since for each of those time slots there are at most d stations scheduled to transmit in that time slot. Notice that an active station might be scheduled to transmit in more than one of those time slots but it will become inactive as soon as it transmits with success. Let $p \leq k$ be the total number of active stations. Since the Boolean sum of the rows in R has Hamming weight at least m, then at least $m - (k - p)$ 1-entries in that Boolean sum are associated with active stations, and consequently, at least $m - (k - p)$ active stations transmit with success and at most $p - (m - (k - p)) = k - m$ active stations do not succeed to transmit their messages.

In the following we will show how to use (k, m, d, n)-selectors to obtain a KG (k, d, n)-code. The idea of this construction is similar to the one employed in [11, 21] to obtain a KG $(k, 1, n)$-code by using (k, m, n)-selectors as building blocks. From now on, unless specified differently, "log" will denote the logarithm in base 2. For the moment, let us assume for the sake of the simplicity that k and d be powers of 2. Our construction works as follows. We concatenate the rows of $(2^{v+1}, 2^v, d, n)$-selectors, for $v = \log d, \ldots, \log k - 1$, with the rows of the $(k, k/2, d, n)$-selector being placed at the top and those of the $(2d, d, d, n)$-selector being placed at the bottom. Then we add an all-1 row at the bottom of the matrix. Let M be the resulting matrix. Notice that the protocol defined by M consists in running the protocols defined by the $(2^{v+1}, 2^v, d, n)$-selectors, starting from the protocol associated with the $(k, k/2, d, n)$-selector through the one associated with the $(2d, d, d, n)$-selector. In the last time slot, corresponding to the bottommost row of M, the protocol schedules all stations to transmit. We will show that M defines a scheduling algorithm the allows up to k active stations to transmit with success, which, by Theorem 4, is equivalent to showing that M is a KG (k, d, n)-code. Let us assume that there are at most $k \leq n$ active stations. We observed that a (k, m, d, n)-selector provides a scheduling algorithm that schedules the transmissions so that at most $k - m$ active stations do not succeed to transmit their messages. Therefore, after running the scheduling protocol for $v = \log k - 1$, i.e., the protocol associated with the $(k, k/2, d, n)$-selector, the algorithm is left with at most $k/2$ active stations. Then the algorithm runs the protocol for $v = \log k - 2$, i.e., the protocol associated with the $(k/2, k/4, d, n)$-selector. This protocol allows all but at most $k/4$ of the remaining active stations to transmit with success. Extending this reasoning to an arbitrary $v \in \{\log d, \ldots, \log k - 1\}$, we have that after running the protocol associated with the $(2^{v+1}, 2^v, d, n)$-selector, there are at most 2^v stations that are still active. Therefore, after running the protocol associated with the $(2d, d, d, n)$-selector, there are most d active stations and no conflict can occur in the last time slot. In the last time slot all stations are scheduled to transmit, and consequently, all remaining active stations transmit with success in that time slot. For arbitrary values of k and d (not necessarily powers of 2), we replace in the above construction $\log k$ and $\log d$ by $\lceil \log k \rceil$ and $\lfloor \log d \rfloor$, respectively. The above construction implies the following upper bound on the minimum length $t_{KG}(k, d, n)$

of a KG (k, d, n)-code:

$$t_{KG}(k, d, n) = O\Big(\sum_{i=\lfloor \log d \rfloor}^{\lceil \log k \rceil - 1} t_{sel}(2^{i+1}, 2^i, d, n)\Big). \tag{1}$$

2.3 Generalized Locally Thin Codes

In this section we define a novel combinatorial structure that is strictly related to our problem in that non-existential results for this structure translate into non-existential results for KG (k, d, n)-codes.

▶ **Definition 6.** Let k, d, and n be integers such that $1 \leq d \leq k \leq n$. A $t \times n$ Boolean matrix M is said to be a $(\leq k, d, n)$-locally thin code of length t if the submatrix formed by any subset of s, $d \leq s \leq k$, columns of M contains a row with a number of 1's comprised between 1 and d. We will denote by $t_{LT}(\leq k, d, n)$ the minimum length of a $(\leq k, d, n)$-locally thin code.

Let M be a $(\leq k, d, n)$-locally thin code and let $\mathbf{F}$ be the family of the sets whose characteristic vectors are the columns of M. The family $\mathbf{F}$ has the property that for any subfamily $\mathbf{F'} \subseteq \mathbf{F}$ with $d \leq |\mathbf{F'}| \leq k$, there exists an element $x \in [t]$ such that $1 \leq |\{F \in \mathbf{F'} : x \in F\}| \leq d$. For $d = 1$, these families correspond to the selective families of [3, 7] and to the $\leq k$-locally thin families of [10]. The authors of [8, 9, 10] proved an $\Omega(k \log(n/k))$ lower bound on the minimum size of the ground set of $\leq k$-locally thin families which is tight with the upper bound on the length of KG $(k, 1, n)$-code [20].

The following theorem establishes a relation between $(\leq k, d, n)$-locally thin codes and KG (k, d, n)-codes.

▶ **Theorem 7.** *Let k, d, and n be integers such that $1 \leq d \leq k \leq n$. Any KG (k, d, n)-code is a $(\leq k, d, n)$-locally thin code.*

Proof. Let M be a KG (k, d, n)-code and suppose by contradiction that M is not a $(\leq k, d, n)$-locally thin code. This implies that there exists a subset of s, $d \leq s \leq k$, columns of M such that the submatrix M_s formed by these s columns contains no row with a number of 1's comprised between 1 and d. Let M' be a k-column submatrix of M such that $M' \supseteq M_s$. Since M is a KG (k, d, n)-code, Definition 3 implies that there exists a non-empty set of row indices $\{i_1, \ldots, i_\ell\} \subseteq [t]$, with $i_1 < i_2 < \ldots < i_\ell$, such that the following property holds:

> There exists a partition $\{M'_1, \ldots, M'_\ell\}$ of the set of columns of M' such that, for $j = 1, \ldots, \ell$, $1 \leq |M'_j| \leq d$ and the i_j-th row of M' has all entries at the intersection with the columns of M'_j equal to 1 and those at the intersection with the columns in $M'_{j+1}, \ldots, M'_\ell$ equal to 0.

Let $M'_{f_1}, \ldots, M'_{f_b} \in \{M'_1, \ldots, M'_\ell\}$, with $f_1 < f_2 < \ldots < f_b$, be the members of the partition having non-empty intersection with M_s, i.e., $M_s \cap M_{f_q} \neq \emptyset$, for each $q \in \{1, \ldots, b\}$, and $M_s \cap (M'_{f_1} \cup \ldots \cup M'_{f_b}) = M_s$. By our assumption that M_s does not contain any row with Hamming weight comprised between 1 and d, it follows that, for each row index $i \in [t]$, the i-th row of M_s has either Hamming weight 0 or Hamming weight larger than d. In the former case, the i-th row has a 0 in correspondence of at least one column in each of $M'_{f_1}, \ldots, M'_{f_b}$, whereas in the latter case the row has entries equal to 1 in correspondence of columns belonging to at least two of $M'_{f_1}, \ldots, M'_{f_b}$, since these submatrices contain at most d columns. Let us consider the row of M_s with index i_{f_1}. By Definition 3, one has that this

row has the entries at the intersection with the columns in M'_{f_1} equal to 1 and those at the intersection with the columns in $M'_{f_2} \cup \ldots \cup M'_{f_b}$ equal to 0. However, from what we have just observed, the i_{f_1}-th row of M_s has either a 0 in correspondence of at least one column in each of $M'_{f_1}, \ldots, M'_{f_b}$, or has entries equal to 1 in correspondence of columns belonging to at least two of $M'_{f_1}, \ldots, M'_{f_b}$. In the former case, the i_{f_1}-th row of M_s has an entry equal to 0 also at the intersection with some column in M'_{f_1}, whereas in the latter case the i_{f_1}-th row has a 1-entry in correspondence of some column in at least one of $M'_{f_2}, \ldots, M'_{f_b}$. In both cases, the i_{f_1}-the row of M_s does not satisfy the property of Definition 3, thus contradicting the fact that M is a KG (k, d, n)-code. ◀

3 Existential Results

First we prove an existential result for (k, m, d, n)-selectors for $k > 2(m-1)$. In order to prove this result, we need to recall the celebrated Lovász Local Lemma for the symmetric case (see [2]), as stated below.

▶ **Lemma 8.** *Let $E_1, E_2, \ldots, E_b$ be events in an arbitrary probability space. Suppose that each event E_i is mutually independent of a set of all other events E_j except for at most D, and that $\Pr[E_i] \leq P$ for all $1 \leq i \leq b$. If $eP(D+1) \leq 1$, then $\Pr[\cap_{i=1}^{n} \bar{E}_i] > 0$.*

By exploiting the above result we prove the following theorem.

▶ **Theorem 9.** *Let k, m, d, and n be positive integers such that $1 \leq d \leq m$ and $2(m-1) < k \leq n$. The minimum size $t_{sel}(k, m, d, n)$ of a (k, m, d, n)-selector is*

$$t_{sel}(k,m,d,n) \leq \begin{cases} 16\left(\ln k + (k-1)\ln\left(\frac{en}{k-1}\right) + (k-m+1)\ln\left(\frac{ek}{k-m+1}\right) + 1\right) & \text{if } 1 \leq d \leq 2 \\ \frac{\ln k + (k-1)\ln\left(\frac{en}{k-1}\right) + (k-m+1)\ln\left(\frac{ek}{k-m+1}\right) + 1}{\frac{d(k-m+1)}{4k} - \ln(4/3)} & \text{otherwise,} \end{cases}$$

where e denotes the Neper's constant $e = 2,71828\ldots$.

Proof. We will prove the existence of a (k, m, d, n)-selector with size smaller than or equal to the stated upper bound. The proof is based on Lemma 8. Let M be a $t \times n$ random binary matrix M where each entry is 1 with probability p and 0 with probability $1-p$. For a given k-column submatrix M' of M, let us denote by $E_{M'}$ the event that M' does not satisfy the property of Definition 5. Notice that, since there are at most $k\binom{n-1}{k-1}$ k-column submatrices containing one or more columns of M', it holds that $E_{M'}$ is independent from all but at most $k\binom{n-1}{k-1}$ events in $\{E_{\tilde{M}} : \tilde{M} \subseteq M, |\tilde{M}| = k\} \setminus \{E_{M'}\}$. In order to apply Lemma 8, we need to derive an upper bound P on the probability of each given event $E_{M'} \in \{E_{\tilde{M}} : \tilde{M} \subseteq M, |\tilde{M}| = k\}$.

In the following we will say that a row is w-good if its Hamming weight is comprised between 1 and d. The probability $Pr\{E_{M'}\}$ is the probability that the submatrix M' contains no subset R of w-good rows such that the Boolean sum of the rows in R has Hamming weight larger than or equal to m. To this aim, we notice that this event holds if and only if there exists a set A of $k-m+1$ column indices such that all w-good rows of M' have all zeros at the intersection with the columns with indices in A. For a fixed subset A of $k-m+1$ indices of columns of M', we denote by $\hat{E}_A$ the event that all w-good rows of M' have zeros at the intersection with the columns with indices in A. Hence, we have that

$$Pr\{E_{M'}\} = Pr\Big\{ \bigcup_{\substack{A \subseteq M': \\ |A| = k-m+1}} \hat{E}_A \Big\} \leq \sum_{\substack{A \subseteq M': \\ |A| = k-m+1}} Pr\{\hat{E}_A\}. \tag{2}$$

For a fixed A, event $\hat{E}_A$ holds if and only if, for any row index $i = 1, \ldots, t$, one has either that the i-th row of M' is not w-good or that the i-th row of M' is w-good and has all zeroes at the intersection with the columns with indices in A. Therefore, one has that

$$\begin{aligned} Pr\{\hat{E}_A\} &= Pr\Big\{\bigcap_{i=1}^{t}\{\{\text{the } i\text{-th row of } M' \text{ is not } w\text{-good}\} \\ &\qquad \cup\{\text{the } i\text{-th row of } M' \text{ is } w\text{-good and } M'(i,j)=0 \text{ for all } j\in A\}\}\Big\} \\ &\leq \prod_{i=1}^{t}\Big(Pr\{\text{the } i\text{-th row of } M' \text{ is not } w\text{-good}\} \\ &\qquad + Pr\{\text{the } i\text{-th row of } M' \text{ is } w\text{-good and } M'(i,j)=0 \text{ for all } j\in A\}\Big) \\ &= (P_1+P_2)^t, \end{aligned} \tag{3}$$

where

$$P_1 = Pr\{\text{the } i\text{-th row of } M' \text{ is not } w\text{-good}\} \tag{4}$$

and

$$P_2 = \{\text{the } i\text{-th row of } M' \text{ is } w\text{-good and } M'(i,j)=0 \text{ for all } j\in A\}, \tag{5}$$

for any fixed $i \in [t]$. Notice indeed that P_1 and P_2 do not depend on i.

By (2) and (3), we have that

$$Pr\{E_{M'}\} \leq \binom{k}{k-m+1}(P_1+P_2)^t. \tag{6}$$

Applying Lemma 8 with $P = \binom{k}{k-m+1}(P_1+P_2)^t$ and $D = k\binom{n-1}{k-1}$, one has that M has positive probability of being a (k,m,n)-strongly selective code if

$$e\binom{k}{k-m+1}(P_1+P_2)^t\left(k\binom{n-1}{k-1}+1\right) \leq 1. \tag{7}$$

Inequality (7) holds for t satisfying the following inequality

$$t \geq \frac{1+\ln\binom{k}{k-m+1}+\ln\left(k\binom{n-1}{k-1}+1\right)}{-\ln(P_1+P_2)}. \tag{8}$$

Therefore, there exists a (k,m,d,n)-selector of size t, for any t satisfying the above inequality.

The proof of the following claim is omitted and will be given in the extended version of the paper.

Claim 1. Let k, m, d, and n be positive integers such that $1 \leq d \leq m$ and $2(m-1) < k \leq n$. If we choose

$$p = \begin{cases} \frac{d}{2k} & \text{if } d \in \{1,2\} \\ \frac{d}{4k} & \text{if } d \geq 3 \end{cases}$$

then it holds

$$-\ln(P_1+P_2) \geq \begin{cases} \frac{1}{16} & \text{if } d \in \{1,2\} \\ (k-m+1)\left(\frac{d}{4k}\right)-\ln\left(\frac{4}{3}\right) & \text{if } d \geq 3. \end{cases}$$

In order for a (k, m, d, n)-selector of size t to exist it is sufficient that t satisfies inequality (8). Claim 1 implies that the righthand side of (8) is at most

$$16\left(1+\ln\binom{k}{k-m+1}+\ln\left(k\binom{n-1}{k-1}+1\right)\right),$$

if $1 \leq d \leq 2$, and it is at most

$$\frac{1+\ln\binom{k}{k-m+1}+\ln\left(k\binom{n-1}{k-1}+1\right)}{\frac{d(k-m+1)}{4k}-\ln(\frac{4}{3})},$$

if $d \geq 3$. Therefore, the upper bounds on $t_{sel}(k, m, d, n)$ in the statement of the theorem are implied by these two upper bounds on the righthand side of (8), along with inequality $\left(k\binom{n-1}{k-1}+1\right) \leq k\binom{n}{k-1}$ and the following well known upper bound on the binomial coefficient:

$$\binom{z}{y} \leq \left(\frac{ez}{y}\right)^y. \qquad \blacktriangleleft$$

Theorem 9 implies that bound (1) on $t_{KG}(k, d, n)$ in Section 2 is

$$O\Big(\sum_{i=\lfloor \log d\rfloor}^{\lceil \log k\rceil -1}\frac{2^{i+1}}{d}\log\frac{n}{2^{i+1}}\Big)=O\left(\frac{k}{d}\log\frac{n}{k}\right).$$

Therefore, the following theorem holds.

▶ **Theorem 10.** *Let k, d, and n be positive integers such that $d \leq k \leq n$. The minimum length $t_{KG}(k, d, n)$ of a KG (k, d, n)-code is*

$$t_{KG}(k, d, n) = O\left(\frac{k}{d}\log\frac{n}{k}\right).$$

Theorem 1 follows from Theorems 4 and 10. In virtue of Theorem 7, we have that Theorem 10 implies an existential results for $(\leq k, d, n)$-locally thin code. For $d = 1$, this existential result attains the same asymptotic upper bound as the one in [8].

Below, we provide a randomized algorithm that generates (k, m, d, n)-selectors meeting the upper bound of Theorem 9. Algorithm 1 is obtained by specializing a technique introduced by Moser and Tardos [23] to generate the structures whose existence is guaranteed by the Lovász Local Lemma. Theorem 1.2 of Moser and Tardos [23] implies that the expected number of times the resampling step (line 14 in Algorithm 1) is repeated is at most $\frac{n}{k^2}$. As a consequence, for fixed k, Algorithm 1 runs in expected polynomial time.

4 Non existential results

The following theorem states a lower bound on the minimum length of $(\leq k, d, n)$-locally thin codes. The proof of this theorem exploits and generalizes an interesting lower bound proof technique used by the authors of [1].

▶ **Theorem 11.** *Let k, d, and n be positive integers such that $3(d + 1) \leq k \leq n$. The minimum length $t_{LT}(\leq k, d, n)$ of a $(\leq k, d, n)$-locally thin code is*

$$t_{LT}(\leq k, d, n) > \frac{\lfloor \frac{k}{d+1}\rfloor}{\log\left(e\lfloor \frac{k}{d+1}\rfloor\right)}\log\left(\frac{n}{k(d+1)}\right).$$

Algorithm 1: Algorithm that generates (k, m, d, n)-selectors.

Input: Integers k, m and n, where $1 \leq d \leq m$ and $2(m-1) < k \leq n$.

Output: M : a (k, m, d, n)-selector.

1 Let $t := \begin{cases} 16\left(\ln k + (k-1)\ln\left(\frac{en}{k-1}\right) + (k-m+1)\ln\left(\frac{ek}{k-m+1}\right) + 1\right) & \text{if } 1 \leq d \leq 2 \\ \frac{\ln k + (k-1)\ln\left(\frac{en}{k-1}\right) + (k-m+1)\ln\left(\frac{ek}{k-m+1}\right) + 1}{\frac{d(k-m+1)}{4k} - \ln(4/3)} & \text{otherwise,} \end{cases}$;

2 Let $p := \begin{cases} \frac{d}{2k} & \text{if } 1 \leq d \leq 2 \\ \frac{d}{4k} & \text{otherwise.} \end{cases}$;

3 Construct a $t \times n$ matrix M where each entry $M(i,j)$ is chosen independently at random with $Pr\{M(i,j) = 1\} = p$ and $Pr\{M(i,j) = 0\} = 1 - p$.;

4 **repeat**

5 Set *flag*:= true;

6 **for** *each set C of k columns of M* **do**

7 **if** *C does not satisfy the property of Definition 5* **then**

8 Set *flag*:= false;

9 Set *missing-column-set* := C;

10 **break**;

11 **end**

12 **end**

13 **if** *flag = false* **then**

14 Choose all the entries in the k columns of *missing-column-set* independently at random, with each of those entries being 1 with probability p and 0 with probability $1 - p$.;

15 **end**

16 **until** *flag = true*;

17 Output M;

Proof. Let us write k as $k = (d+1)\lfloor\frac{k}{d+1}\rfloor + q$, with $0 \leq q \leq d$ and let $u = \lfloor\frac{k}{d+1}\rfloor$. We denote by α a positive rational number $\alpha = \frac{a}{b}$ satisfying the following inequalities

$$\frac{1}{u} \leq \alpha < \frac{1}{2}. \tag{9}$$

Let us denote by $n_{LT}(\leq k, d, t)$ the maximum value of n for which there exists a $(\leq k, d, n)$-locally thin code of length t. We will prove that

$$n_{LT}(\leq k, d, t) < k(d+1) \cdot 2^{h(\alpha)t}. \tag{10}$$

First we will show that for any $\alpha < \frac{1}{2}$ it holds

$$\frac{\alpha}{e} \leq 2^{-\frac{h(\alpha)}{\alpha}} < \frac{\alpha}{2}, \tag{11}$$

where $h(\alpha)$ denotes the binary entropy of α. Notice that, since we can choose $\alpha = \frac{1}{u}$, the upper bound (10) on $n_{LT}(\leq k, d, t)$, along with the lefthand side of (11), implies that

$$n_{LT}(\leq k, d, t) < k(d+1)\left(e\lfloor\frac{k}{d+1}\rfloor\right)^{t/\lfloor\frac{k}{d+1}\rfloor},$$

from which the lower bound on $t_{LT}(\leq k, d, n)$ in the statement of the theorem follows.

Let us prove inequalities (11). By the definition of binary entropy, one has that

$$h\left(\frac{a}{b}\right) = \frac{a}{b}\log\frac{b}{a} + \left(\frac{b-a}{b}\right)\log\left(\frac{b}{b-a}\right) = \frac{a}{b}\log\frac{b}{a} + \frac{1}{b}\cdot\log\left(1+\frac{a}{b-a}\right)^{b-a}. \quad (12)$$

Since $\left(1+\frac{a}{b-a}\right)^{b-a}$ increases with b, one has that $2^a < \left(1+\frac{a}{b-a}\right)^{b-a} \leq e^a$, where the left inequality follows from the righthand side of (9) that implies $b > 2a$. Therefore, by (12), it holds

$$\frac{a}{b}\log\left(\frac{2b}{a}\right) < h\left(\frac{a}{b}\right) \leq \frac{a}{b}\log\left(\frac{eb}{a}\right). \quad (13)$$

By replacing $\frac{a}{b}$ with α, inequalities (13) can be rewritten as $\alpha\log\left(\frac{2}{\alpha}\right) < h(\alpha) \leq \alpha\log\left(\frac{e}{\alpha}\right)$, from which we have that inequalities (11) hold.

Now we prove that $n_{LT}(\leq k, d, t) < k(d+1)\cdot 2^{h(\alpha)t}$. The proof is by induction on t.

For $t = 1$, any $t \times n$ Boolean matrix M has a single row that either contains at least $\frac{n}{2}$ entries equal to 0 or at least $\frac{n}{2}$ entries equal to 1. Consequently, if we assume by contradiction that $|M| = n \geq k(d+1)\cdot 2^{h(\alpha)t} \geq k(d+1)$ then the single row of M would either contain at least $k(d+1)/2$ occurrences of 0 or at least $k(d+1)/2$ occurrences of 1. This implies that there exist $k(d+1)/2 \geq k$ entries that are either all equal to 0 or all equal to 1 thus contradicting the hypothesis that M is a $(\leq k, d, n)$-locally thin code.

Let us consider $t > 1$ and let us assume by induction hypothesis that $n_{LT}(\leq k, d, t-1) < k(d+1)\cdot 2^{h(\alpha)(t-1)}$. Let M be a $t \times n$ $(\leq k, d, n)$-locally thin code of length t and let us assume by contradiction that $n \geq k(d+1)\cdot 2^{h(\alpha)t}$. We consider the following two cases.

Case 1. There exists an integer i in $[t]$ such that there are at least $2^{-h(\alpha)}n$ columns of M with the i-th entry equal to 0. In this case, if we remove the i-th entry from each of these columns, we have that the resulting columns form a matrix $\tilde{M}$ that is a $(\leq k, d, n)$-locally thin code of length $t-1$. Since we are assuming that $n \geq k(d+1)\cdot 2^{h(\alpha)t}$, it holds $|\tilde{M}| \geq 2^{-h(\alpha)}k(d+1)2^{h(\alpha)t} = k(d+1)\cdot 2^{h(\alpha)(t-1)}$. By induction hypothesis, $\tilde{M}$ cannot be a $(\leq k, d, n)$-locally thin code of length $t-1$, thus contradicting the fact that M is $(\leq k, d, n)$-locally thin code.

Case 2. For each element $i \in [t]$, there are less than $2^{-h(\alpha)}n$ columns of M with the i-th entry equal to 0. This implies that for a fixed i and for u randomly chosen columns $\mathbf{c}_1, \ldots, \mathbf{c}_u$ of M, the probability that $\mathbf{c}_1, \ldots, \mathbf{c}_u$ all have the i-th entry equal to 0 is less than $2^{-uh(\alpha)}$. By the lefthand side of (9) this probability is at most $2^{-\frac{h(\alpha)}{\alpha}}$, which by the righthand side of (11) is less than $\frac{\alpha}{2}$. Therefore, the expected number of 0-entries in the Boolean sum $\bigvee_{j=1}^{u}\mathbf{c}_j$ is less than $\frac{t\alpha}{2}$. Let X denote the number of 0-entries in the Boolean sum of u randomly chosen columns. We have shown that $E[X] < \frac{t\alpha}{2}$. Markov's inequality implies that, for any non-negative random variable Y and for any $b > 0$, it holds $Pr\{Y \geq b\} \leq \frac{E[Y]}{b}$. By our upper bound on $E[X]$ and by Markov's inequality, one has $\Pr\{\bigvee_{j=1}^{u}\mathbf{c}_j$ has at least $t\alpha$ 0-entries$\} < \frac{t\alpha}{2}\cdot\frac{1}{t\alpha} = \frac{1}{2}$. It follows that $Pr\{\bigvee_{j=1}^{u}\mathbf{c}_j$ has Hamming weight larger than $t - t\alpha\} > \frac{1}{2}$. Let $m = 2(d+1)\lceil 2^{h(\alpha)t}\rceil$ and let $\mathcal{B}_1, \ldots, \mathcal{B}_m$ be m randomly chosen subsets of u columns of M such that $\mathcal{B}_j \cap \mathcal{B}_\ell = \emptyset$, for $j \neq \ell$. Such subsets $\mathcal{B}_1, \ldots, \mathcal{B}_m$ can be generated by randomly permuting the columns of M, and then picking a set of $m \cdot u$ consecutive columns in the resulting matrix. In order to obtain $\mathcal{B}_1, \ldots, \mathcal{B}_m$, this set of columns is partitioned into m disjoint subsets each consisting of u consecutive columns. We have shown that $\bigvee_{\mathbf{c}\in\mathcal{B}_\ell}\mathbf{c}$ has Hamming weight larger than $t - t\alpha$ with probability larger than $\frac{1}{2}$, and consequently, the expected number

of subfamilies $\mathcal{B}_j$'s among $\mathcal{B}_1, \ldots, \mathcal{B}_m$ such that $\bigvee_{F \in \mathcal{B}_j} F$ has Hamming weight larger than or equal to $t - t\alpha$ is at least $\frac{m}{2}$. By linearity of expectation, there is a random choice of $\mathcal{B}_1, \ldots, \mathcal{B}_m$ such that there are at least $f \geq \frac{m}{2}$ subfamilies $\mathcal{B}'_1, \ldots, \mathcal{B}'_f$ among $\mathcal{B}_1, \ldots, \mathcal{B}_m$ for which one has that $\bigvee_{\mathbf{c} \in \mathcal{B}'_\ell} \mathbf{c}$, for $\ell = 1, \ldots, f$, has Hamming weight larger than or equal to $t - t\alpha$. However, one has that the number of pairwise distinct binary vector of length t with Hamming weight larger than or equal to $t - t\alpha$ is

$$\sum_{s=t-t\alpha}^{t} \binom{t}{s} = \sum_{s=0}^{t\alpha} \binom{t}{s} \leq 2^{th(\alpha)}, \tag{14}$$

where the last inequality follows from the well known inequality $\sum_{i=0}^{b} \binom{g}{i} \leq 2^{gh(b/g)}$, holding for $b/g \leq 1/2$, [15] . Since it is $m = 2(d+1)\lceil 2^{h(\alpha)t} \rceil$, then there are at most $\frac{m}{2(d+1)}$ pairwise distinct vectors of Hamming weight larger than or equal to $t - t\alpha$. We have shown that there exist $f \geq \frac{m}{2}$ subfamilies $\mathcal{B}'_1, \ldots, \mathcal{B}'_f$ such that $\bigvee_{\mathbf{c} \in \mathcal{B}'_\ell} \mathbf{c}$, for $\ell = 1, \ldots, f$, has Hamming weight larger than or equal to $t - t\alpha$. As a consequence, for at least a binary vector $\mathbf{c}_v$, there are $d+1$ sets $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}} \subseteq \{\mathcal{B}'_1, \ldots, \mathcal{B}'_f\}$ such that $\bigvee_{\mathbf{c} \in \mathcal{B}'_{j_q}} \mathbf{c} = \mathbf{c}_v$, for $q = 1, \ldots, d+1$. In other words, $\mathbf{c}_v$ occurs at least $d+1$ times among the Boolean sums $\bigvee_{\mathbf{c} \in \mathcal{B}'_1} \mathbf{c}, \ldots, \bigvee_{\mathbf{c} \in \mathcal{B}'_f} \mathbf{c}$. Therefore, the submatrix formed by the $(d+1)u = (d+1)\lfloor \frac{k}{d+1} \rfloor \leq k$ columns of $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$ is such that each row is either an all-zero row or has at least $d+1$ entries equal to 1, thus contradicting the assumption the M is a $(\leq k, d, n)$-locally thin code. ◀

Theorem 2 is an immediate consequence of Theorems 7 and 11.

The technique used to prove the lower bound of Theorem 11 allows also to obtain a lower bound on the length of codes satisfying a weaker property than that of $(\leq k, d, n)$-locally thin codes. We refer to these codes as (k, d, n)-*locally thin codes*. A $t \times n$ Boolean matrix M is a (k, d, n)-locally thin code of length t if and only if any submatrix formed by *exactly* k columns of M contains at least a row whose Hamming weight is comprised between 1 and d. If we interpret the columns of such a code as the characteristic vectors of n sets on the ground set $[t]$, then these sets have the property that for any k of them there exists an $i \in [t]$ that is contained in at at least one of these k sets and in no more than d of them. For $d = 1$, these families correspond to the k-locally thin code of [1].

▶ **Theorem 12.** *Let k, d, and n be positive integers such that $4(d+1) \leq k \leq n$. The minimum length $t_{LT}(k, d, n)$ of a (k, d, n)-locally thin code is*

$$t_{LT}(k, d, n) > \frac{\left(\lfloor \frac{k}{d+1} \rfloor - 1\right)}{\log\left(e\left(\lfloor \frac{k}{d+1} \rfloor - 1\right)\right)} \log\left(\frac{n}{k(d+1)}\right).$$

Proof. The proof is similar to that of Theorem 11 with the difference that here we write k as $k = (d+1)\lfloor \frac{k}{d+1} \rfloor + q = (d+1)\left(\lfloor \frac{k}{d+1} \rfloor - 1\right) + d + 1 + q$, with $0 \leq q \leq d$, and set $u = \lfloor \frac{k}{d+1} \rfloor - 1$. Moreover, here in order to prove the lower bound for Case 2, we need to prove the existence of a submatrix of *exactly* k columns that does not contain any row of Hamming weight comprised between 1 and d. To this aim, let us consider the subsets of u columns $\mathcal{B}'_{j_1}, \ldots, \mathcal{B}'_{j_{d+1}}$ whose existence has been proved in the proof of Theorem 11. The subsets $\mathcal{B}'_{j_1}, \ldots, \mathcal{B}'_{j_{d+1}}$ are such that the Boolean sums $\bigvee_{\mathbf{c} \in \mathcal{B}'_{j_1}} \mathbf{c}, \ldots, \bigvee_{\mathbf{c} \in \mathcal{B}'_{j_{d+1}}} \mathbf{c}$ have Hamming weight at least $t - t\alpha$, and the submatrix formed by the columns in $\mathcal{B}'_{j_1} \cup \ldots \cup \mathcal{B}'_{j_{d+1}}$ contains no row of Hamming weight comprised between 1 and d. The number of columns in this submatrix is $(d+1)u = (d+1)\left(\lfloor \frac{k}{d+1} \rfloor - 1\right) \leq k$. We will show that it is possible to add

columns to this submatrix so as to obtain a submatrix with exactly k-columns and with no row of Hamming weight comprised between 1 and d. To this aim, let us consider the columns of M that do not belong to any of $\mathcal{B}'_{j_1}, \ldots, \mathcal{B}'_{j_{d+1}}$. Let us denote by $i_1, \ldots, i_z$ the indices of the 0-zero entries in the Boolean sums $\bigvee_{\mathbf{c}\in\mathcal{B}'_{j_1}} \mathbf{c}, \ldots, \bigvee_{\mathbf{c}\in\mathcal{B}'_{j_{d+1}}} \mathbf{c}$. By the way $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$ have been defined in the proof of Theorem 11, one has $z \leq t\alpha$. We will prove that there are at least $d+1+q$ columns whose restrictions to the entries with indices $i_1, \ldots, i_z$ are identical. This implies that the $t \times k$ submatrix formed by $d+1+q$ of these columns and the columns in $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$ is such that each row is either an all-zero row or has at least $d+1$ entries equal to 1, thus contradicting the fact that M is a $(\leq k, d, n)$-locally thin code.

The rest of the proof is devoted to prove that there are at least $d+1+q$ distinct columns of M not in $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$ whose restrictions to the entries with indices $i_1, \ldots, i_z$ are identical. We observe that the number of columns of M that do not belong to any of $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$ is $n-(d+1)\left(\lfloor\frac{k}{d+1}\rfloor - 1\right)$. By the contradiction assumption, it holds $n \geq k(d+1)\cdot 2^{h(\alpha)t}$, and consequently, the above said number of columns is at least $k(d+1)\cdot 2^{h(\alpha)t} - (d+1)\left(\lfloor\frac{k}{d+1}\rfloor - 1\right)$ which, by the righthand side of (11), is larger than $k(d+1)\left(\frac{2}{\alpha}\right)^{t\alpha} - (d+1)\left(\lfloor\frac{k}{d+1}\rfloor - 1\right)$. Since $k(d+1)\cdot\left(\frac{2}{\alpha}\right)^{t\alpha} - (d+1)\left(\lfloor\frac{k}{d+1}\rfloor - 1\right) > kd\cdot\left(\frac{2}{\alpha}\right)^{t\alpha} > 2d\cdot 2^{t\alpha}$, it follows that there are more than $2d\cdot 2^{t\alpha}$ columns of M not in $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$. Among these columns there are at most $2^z \leq 2^{t\alpha}$ columns whose restrictions to indices $i_1, \ldots, i_z$ are pairwise distinct. As a consequence, there are at least $2d+1 \geq d+1+q$ columns of M not in $\mathcal{B}'_{j_1} \ldots, \mathcal{B}'_{j_{d+1}}$ whose restrictions to indices $i_1 \ldots, i_z$ are identical. ◀

For k even, the authors of [1] proved an $\Omega(k \log n)$ lower bound on the minimum size of the ground set of k-locally thin families, whereas, for arbitrary values of k, they gave an $\Omega\left(\frac{k}{\log k}\log n\right)$ lower bound. This latter lower bound is asymptotically the same as the lower bound obtained by setting d equal to 1 in the lower bound of Theorem 12.

Notice that Theorem 12 gives a lower bound on the minimum number of time slots needed to solve all conflicts in our model when the number of active stations is *exactly* k.

5 Conclusions

We have presented upper and lower bounds on the minimum number of time slots needed to solve conflicts among up to k active stations in a multiple-access system with feedback where at most d stations can transmit simultaneously with success over the channel. Interestingly, we have proved that it is possible to resolve conflicts in a number of time slots linearly decreasing with the number d of messages that can be simultaneously transmitted with success. Indeed, we have provided a conflict resolution algorithm that uses a $1/d$ ratio of the number of time slots used by the optimal conflict resolution algorithm for the particular case $d = 1$ [20].

The upper and lower bounds given in this paper differ asymptotically by a $\log(k/d)$ factor. An interesting open problem is to close this gap by improving on the lower bound on the minimum length of KG $(\leq k, d, n)$-codes.

References

1 Noga Alon, Emanuela Fachini, and János Körner. Locally thin set families. *Combinatorics, Probability and Computing*, 9(06):481–488, 2000.

2 Noga Alon and Joel Spencer. *The Probabilistic Method. Interscience series in discrete mathematics and optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., Hoboken, NJ„ third edition, 2008.

3 Stefano Basagni, Danilo Bruschi, and Imrich Chlamtac. A mobility-transparent deterministic broadcast mechanism for ad hoc networks. *IEEE/ACM transactions on networking*, 7(6):799–807, 1999.

4 Keren Censor-Hillel, Bernhard Haeupler, Nancy Lynch, and Muriel Médard. Bounded-contention coding for the additive network model. *Distributed Computing*, 28(5):297–308, 2015.

5 Douglas S. Chan, Toby Berger, and Lang Tong. Carrier sense multiple access communications on multipacket reception channels: theory and applications to IEEE 802.11 wireless networks. *IEEE Transactions on Communications*, 61(1):266–278, 2013.

6 Bogdan S. Chlebus. Randomized communication in radio networks. In P. M. Pardalos, S. Rajasekaran, J. Reif, and J. D. P. Rolim, editors, *Handbook of Randomized Computing*, volume 1, pages 401–456. Kluwer Academic Publishers, 2001.

7 Bogdan S. Chlebus, Leszek Gąsieniec, Alan Gibbons, Andrzej Pelc, and Wojciech Rytter. Deterministic broadcasting in unknown radio networks. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'00, pages 861–870, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

8 Andrea E. F. Clementi, Angelo Monti, and Riccardo Silvestri. Selective families, superimposed codes, and broadcasting on unknown radio networks. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 709–718. Society for Industrial and Applied Mathematics, 2001.

9 Gérard D. Cohen. Applications of coding theory to communication combinatorial problems. *Discrete Mathematics*, 83(2):237–248, 1990.

10 Miklós Csűrös and Miklós Ruszinkó. Single-user tracing and disjointly superimposed codes. *IEEE transactions on information theory*, 51(4):1606–1611, 2005.

11 Annalisa De Bonis, Leszek Gąsieniec, and Ugo Vaccaro. Optimal two-stage algorithms for group testing problems. *SIAM Journal on Computing*, 34(5):1253–1270, 2005.

12 Annalisa De Bonis and Ugo Vaccaro. Constructions of generalized superimposed codes with applications to group testing and conflict resolution in multiple access channels. *Theoretical Computer Science*, 306(1):223–243, 2003.

13 Annalisa De Bonis and Ugo Vaccaro. Optimal algorithms for two group testing problems, and new bounds on generalized superimposed codes. *IEEE transactions on information theory*, 52(10):4673–4680, 2006.

14 Aditya Dua. Random access with multi-packet reception. *IEEE Transactions on Wireless Communications*, 7(6):2280–2288, 2008.

15 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory.* Springer, 2006.

16 Sylvie Ghez, Sergio Verdu, and Stuart C. Schwartz. Stability properties of slotted aloha with multipacket reception capability. *IEEE Transactions on Automatic Control*, 33(7):640–649, 1988.

17 Sylvie Ghez, Sergio Verdú, and Stuart C. Schwartz. Optimal decentralized control in the random access multipacket channel. *IEEE Transactions on Automatic Control*, 34(11):1153–1163, 1989.

18 Jasper Goseling, Michael Gastpar, and Jos H. Weber. Random access with physical-layer network coding. *IEEE Transactions on Information Theory*, 61(7):3670–3681, 2015.

19 Albert G. Greenberg and Schmuel Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *Journal of the ACM (JACM)*, 32(3):589–596, 1985.

20 Janos Komlos and Albert Greenberg. An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, 1985.

21 Dariusz R. Kowalski. On selection problem in radio networks. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 158–166. ACM, 2005.

22 Avery Miller. On the complexity of neighbourhood learning in radio networks. *Theoretical Computer Science*, 608:135–145, 2015.

23 Robin A. Moser and Gábor Tardos. A constructive proof of the general lovász local lemma. *Journal of the ACM (J. ACM)*, 57(2):11–15, 2010.

24 Alexander Russell, Sudarshan Vasudevan, Bing Wang, Wei Zeng, Xian Chen, and Wei Wei. Neighbor discovery in wireless networks with multipacket reception. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1984–1998, 2015.

25 Boris Tsybakov. Packet multiple access for channel with binary feedback, capture, and multiple reception. *IEEE Transactions on Information Theory*, 50(6):1073–1085, 2004.

How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses

Aras Atalar[1], Paul Renaud-Goud[2], and Philippas Tsigas[3]

1 **Chalmers University of Technology, Göteborg, Sweden**
`aaras@chalmers.se`
2 **Toulouse Institute of Computer Science Research, Toulouse, France**
`prenaud@irit.fr`
3 **Chalmers University of Technology, Göteborg, Sweden**
`tsigas@chalmers.se`

Abstract

In this paper we present two analytical frameworks for calculating the performance of lock-free data structures. Lock-free data structures are based on retry loops and are called by application-specific routines. In contrast to previous work, we consider in this paper lock-free data structures in dynamic environments. The size of each of the retry loops, and the size of the application routines invoked in between, are not constant but may change dynamically. The new frameworks follow two different approaches. The first framework, the simplest one, is based on queuing theory. It introduces an average-based approach that facilitates a more coarse-grained analysis, with the benefit of being ignorant of size distributions. Because of this independence from the distribution nature it covers a set of complicated designs. The second approach, instantiated with an exponential distribution for the size of the application routines, uses Markov chains, and is tighter because it constructs stochastically the execution, step by step.

Both frameworks provide a performance estimate which is close to what we observe in practice. We have validated our analysis on

(i) several fundamental lock-free data structures such as stacks, queues, deques and counters, some of them employing helping mechanisms, and
(ii) synthetic tests covering a wide range of possible lock-free designs.

We show the applicability of our results by introducing new back-off mechanisms, tested in application contexts, and by designing an efficient memory management scheme that typical lock-free algorithms can utilize.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Lock-free, Data Structures, Parallel Computing, Performance, Modeling, Analysis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.23

1 Introduction

During the last two decades, lock-free data structures have received a lot of attention in the literature, and have been accepted in industrial applications, *e.g.* in the Intel's Threading Building Blocks Framework [12], the Java concurrency package [20] and the Microsoft .NET Framework [18].

Naturally, the development of lock-free data structures was accompanied by studies on the performance of such data structures, in order to characterize their scalability. Having no guarantee on the execution time of an individual operation, the time complexity analyses of lock-free algorithms have turned towards amortized analyses. The so-called amortized analyses

are thus interested in the worst-case behavior over a sequence of operations, which can be seen as a worst-case bound on the average time per operation. In order to cover various contention environments, the time complexity of the algorithms is often parameterized by different contention measures, such as point [5], interval [1] or step [6] contention. Nonetheless these investigations are targeting worst-case asymptotic behaviors. There is a lack of analytical results in the literature capable of describing the execution of lock-free algorithms on top of a hardware platform, and providing predictions that are close to what is observed in practice. Asymptotic bounds are particularly useful to rank different algorithms, since they rely on a strong theoretical background, but the presence of potentially high constants might produce misleading results. Yet, an absolute prediction of the performance can be of great importance by constituting the first step for further optimizations.

The common measure of performance for data structures is throughput, defined as the number of operations on the data structure per unit of time. To this end, this performance measure is usually obtained by considering an algorithm that strings together a pure sequence of calls to an operation on the data structure. However, when used in a more realistic context, the calls to the operations are mixed with application-specific code (that we call here parallel work). For instance, in a work-stealing environment designed with deques, a thread basically runs one of the following actions: pushing a new-generated task in its deque, popping a task from a deque or executing a task. The modifications on the deques are thus interleaved with deque-independent work. There exist some papers that consider in their experiments local computations between calls to operations during their respective evaluations, but the amount of local computations follows a given distribution varying from paper to paper, *e.g.* constant [17], uniform [10], exponential [22].

In this work, we derive a general approach for unknown distributions of the size of the application-specific code, as well as a tighter method when it follows an exponential distribution.

As for modeling the data structure itself, we use as a basis the universal construction described by Herlihy in [11], where it is shown that any abstract data type can get such a lock-free implementation, which relies on one retry loop. Moreover, we have particularly focused our experiments on data structures that present a low level of disjoint-access parallelism [13] (stack, queue, shared counter, deque). Coming back to amortized analyses, the time complexity of an operation is often expressed as a contention-free time complexity added with a contention overhead. In this paper, we want to model and analyze the impact of contention. Loosely speaking, the data structures that exhibit low level of disjoint-access parallelism have lightweight operations (*i.e.* low contention-free complexity) and they are prone to high contention overheads. In contrast, the data structures that present high level of disjoint-access parallelism, or that employ contention alleviation techniques, provide heavyweight operations (*i.e.* high contention-free complexity) and behave differently, compared to the previous ones, under contention. Our analyses examine this trade-off and then facilitate conscious decisions in the data structures design and use.

We propose two different approaches that analyze the performance of such data structures. On the one hand, we derive an average-based approach invoking queuing theory, which provides the throughput of a lock-free algorithm without any knowledge about the distribution of the parallel work. This approach is flexible but allows only a coarse-grained analysis, and hence a partial knowledge of the contention that stresses the data structure. On the other hand, we exhibit a detailed picture of the execution of the algorithm when the parallel work is instantiated with an exponential distribution, through a second complementary approach. We prove that the multi-threaded execution follows a Markovian process and a Markov chain

analysis allows us to pursue and reconstruct the execution, and to compute a more accurate throughput.

We finally show several ways to use our analyses and we evaluate the validity of our ideas by experimental results. Those two analysis approaches give a good understanding of the phenomena that drive the performance of a lock-free data structure, at a high-level for the average-based approach, and at a detailed level for the constructive method. Moreover, our results provide a common framework to compare different implementations of a data structure, in a fair manner. We also emphasize that there exist several concrete paths to apply our analyses. To this end, based on the knowledge about the application at hand, we implement two back-off strategies. We show the applicability of these strategies by tuning a Delaunay triangulation application [9] and a streaming pipeline component which is fed with trade exchange workloads [19]. These experiments reveal the validity of our analyses in the application domain, under non-synthetic workloads and diverse access patterns. We confirm the benefits of our theoretical results by designing a new adaptive memory management mechanism for lock-free data structures in dynamic environments which surpasses the traditional scheme and which is such that the loss in performance, when compared to a static data structure without memory management, is largely leveraged. This memory management mechanism is based on the analyses presented in this paper.

2 Related Work

Alistarh *et al.* [2] have studied the same class of lock-free data structures that we consider in this paper. They show initially that the lock-free algorithms are statistically wait-free and going further they exhibit upper bounds on the performance. Their analysis is done in terms of scheduler steps, in a system where only one thread can be scheduled (and can then run) at each step. If compared with execution time, this is particularly appropriate to a system where the instructions of the threads cannot be done in parallel (*e.g.* multi-threaded program on a multi-core processor with only writes on the same cache line of the shared memory). In our paper, the execution is evaluated in terms of processor cycles, strongly related to the execution time. In addition, the "parallel work" and the "critical work" can be done in parallel. Also, in our paper we estimate the throughput (close to the inverse of system latency) for any number of threads.

Comparing to our previous work: In [3], we illustrate the performance impacting factors and the model we use to cover a subset of lock-free structures that we consider in this paper. In the former paper, the analysis is built upon properties that arise only when the sizes of the critical work and the parallel work are *constant*. There, we show that the execution is not memoryless due to the natural synchrony provided by the retry loops; at the end of the line, we prove that the execution is cyclic and use this property to bound the rate of failed retries.

Here, we provide two new approaches which serve different purposes. In the first approach, we relax the assumptions regarding the critical work and parallel work parameters, that we respectively use to model the data structure operations and the application specific code from which the data structure operations are called. The first approach relies on the expected values of the size of the critical work and the parallel work. This allows us to cover, compared to our previous analysis, more advanced lock-free data structure operations, see Section 6.3. Also, we can analyse the data structures running on a larger variety of application specific environments, thanks to the relaxed assumption on the size of the parallel work. The second approach provides a tight analysis when the parallel work follows an exponential distribution.

We can observe a significant decrease in the performance when the parallel work is initiated with exponential distribution in comparison to the cases where the parallel work is constant as in our previous work, see [4]. The tight analyses, in our previous work and the second approach presented in this paper, reveal for the first time an analytical understanding of this phenomenon.

This paper is complementary to the previous work, not only because of the difference in the analysis tools, the extensive set of data structures and the application specific environments that it considers but also because they together exhibit the impact of the size distributions of the parallel work on the performance of lock-free data structures.

3 Preliminaries

We describe in this section the structure of the algorithm that is covered by our model. We explain how to analyze the execution of an instance of such an algorithm when executed by several threads, by slicing this execution into a sequence of adjacent success periods, where a success period is an interval of time during which exactly one operation returns. Each of the success periods is further split into two by the first access to the data structure in the considered retry loop. This execution pattern reflects fundamental phases of both analyses, whose first steps and general direction are outlined at the end of the section.

3.1 System Settings

All threads call Procedure AbstractAlgorithm (see Figure 1) when they are spawned. So each thread follows a simple though expressive pattern: a sequence of calls to an operation on the data structure, interleaved with some parallel work during which the thread does not try to modify the data structure. For instance, it can represent a work-stealing algorithm, as described in the introduction.

The algorithm is decomposed in two main sections: the *parallel section*, represented on line 2, and the *retry loop* (which represents one operation on the shared data structure) from line 3 to line 6. A *retry* starts at line 4 and ends at line 6. The outer loop that goes from line 1 to line 6 is designated as the *work loop*. In each retry, a thread tries to modify the data structure and does not exit the retry loop until it has successfully modified the data structure. The retry loop is composed of at least one retry (and the first iteration of the retry loop is strictly speaking not a retry, but a try).

We denote by cc the execution time of a CAS when the executing thread does not own the cache line in exclusive mode, in a setting where all threads share a last level cache. Typically, there exists a thread that touches the data between two requests of the same thread, therefore this cost is paid at every occurrence of a CAS. As for the *Read*s, rc holds for the execution time of a cache miss. When a thread executes a failed CAS, it immediately reads the same cache line (at the beginning of the next retry), so the cache line is not missing, and the execution time of the *Read* is considered as null. However, when the thread comes back from the parallel section, a cache miss is paid. To conclude with the parameters related to the platform, we use P cores, where the CAS (resp. the *Read*) latency is identical for all cores, *i.e.* cc (resp. rc) is constant.

The algorithm is parameterized by two execution times. In the general case, the execution time of an occurrence of the parallel section (application-specific section) is a random variable that follows an unknown probability distribution. In the same way, the execution time of the critical work (specific to a data structure) can vary while following an unknown probability distribution. The only provided information is the mean value of those two execution times:

```
Procedure AbstractAlgorithm
1 while ! done do
2     Parallel_Work();
3     while ! success do
4         current ← Read(AP);
5         new ← Critical_Work(current);
6         success ← CAS(AP, current, new);
```

Figure 1 Thread procedure.

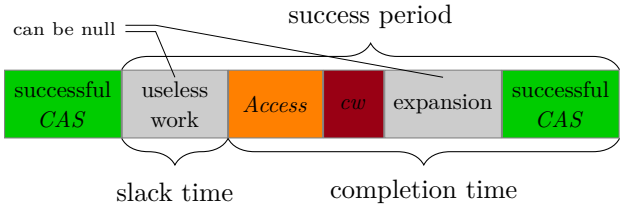


Figure 2 Success period.

cw for the critical work, and *pw* for the parallel work. These values will be given in units of work, where 1 u.o.w. = 50 cycles.

3.2 Execution Description

It has been underlined in [3] that there are two main conflicts that degrade the performance of the data structures which do not offer a great degree of disjoint-access parallelism: logical and hardware conflicts.

Logical conflicts occur when there are more than one thread in the retry loop at a given time (happens typically when the number of threads is high or when the parallel section is small). At any time, considering only the threads that are in the retry loop, there is indeed at most one thread whose retry will be successful (*i.e.* whose ending *CAS* will succeed), which implies the execution of more retries for the failing threads. In addition, after a thread executes successfully its final *CAS*, the other threads of the retry loop have first to finish their current retry before starting a potentially successful retry, since they are not informed yet that their current retry is doomed to failure. This creates some "holes" in the execution where all threads are executing useless work.

The threads will also experience *hardware conflicts*: if several threads are requesting the same data, so that they can operate a *CAS* on it, a single thread will be satisfied. All the other threads will have to wait until the current *CAS* is finished, and give a new try when this *CAS* is done. While waiting for the ownership of the cache line, the requesting threads cannot perform any useful work. This waiting time is referred to as *expansion*.

We now refine the description of the execution of the algorithm. The timeline is initially decomposed into a sequence of success periods that will define the throughput. A success period is an interval of time of the execution that

(i) starts after a successful *CAS*,
(ii) contains a single successful *CAS*,
(iii) finishes after this successful *CAS*.

To be successful in its retry, a thread has first to access the data structure, then modify it locally, and finally execute a *CAS*, while no other thread performs changes on the data structure. That is why each success period is further cut into two main phases (see Figure 2). During the first phase, whose duration is called the *slack time*, no thread is accessing the data structure. The second phase, characterized by the *completion time*, starts with the first access to the data structure (by any thread). Note that this *Access* could be either a *Read* (if the concerned thread just exited the parallel section) or a failed *CAS* (if the thread was already in the retry loop). The next successful *CAS* will come at least after *cw* (one thread has to traverse the critical work anyway), that is why we split the latter phase into: *cw*, then expansion, and finally a successful *CAS*.

3.3 Our Approaches

In this work, we propose two different approaches to compute the throughput of a lock-free algorithm, which we name as average-based and constructive. The average-based approach relies on queuing theory and is focused on the average behavior of the algorithm: the throughput is obtained through the computation of the expectation of the success period at a random time. As for the constructive approach, it describes precisely the instants of accesses and modifications to the data structure in each success period: in this way, we are able to deconstruct and reconstruct the execution, according to observed events. The constructive approach leads to a more accurate prediction at the expense of requiring more information about the algorithm: the distribution functions of the critical and parallel works have indeed to be instantiated.

In both cases, we partition the domain space into different levels of contention (or *modes*); these partitions are independent across approaches, even if we expect similarities, but in each case, cover the whole domain space (all values of critical work, parallel work and number of threads).

3.3.1 Average-based Analysis

We distinguish two main modes in which the algorithm can run: contended and non-contended. In the non-contended mode, *i.e.* when the parallel work is large or the number of threads is low, concurrent operations are not likely to collide. So every retry loop will count a single retry, and atomic primitives will not delay each other. In the contended mode, any operation is likely to experience unsuccessful retries before succeeding (logical conflicts), and a retry will last longer than in the non-contended mode because of the collision of atomic primitives (hardware conflicts).

Once all the parameters are given, the analysis is centered around the calculation of a single variable $\overline{P_{rl}}$, which represents the expectation of the number of threads inside the retry loop at a random instant. Based on this variable, we are able to express the expected expansion $\overline{e}\left(\overline{P_{rl}}\right)$ at a random time. As a next step, we show how this expansion can be used to estimate the expected slack time $\overline{st}\left(\overline{P_{rl}}\right)$ and the expected completion time $\overline{ct}\left(\overline{P_{rl}}\right)$, and at the end, the expected time of a success period $\overline{sp}\left(\overline{P_{rl}}\right)$.

3.3.2 Constructive Method

The previous average-based reasoning is founded on expected values at a random time, while in the constructive approach, we study each success period individually, based on the number of threads at the beginning of the considered success period. So we are able to exhibit more clearly the instants of occurrences of the different accesses and modifications to the data structure, and thus to predict the throughput more accurately.

We rely on the same set of values used in the average-based approach, but these values are now associated with a given success period. Thus the number of threads inside the retry loop P_{rl}, as well as the slack time and the completion time are evaluated at the beginning of each success period. We denote these times in the same way as in the first approach, but remove the bar on top since these values are not expectations any more.

The different contention modes do not characterize here the steady-state of the data structure as in the previous approach but are associated with the current success period. Accordingly, the contention can oscillate through different modes in the course of the execution. First, a success period is not contended when $P_{rl} = 0$, *i.e.* when there is no thread in the retry loop after a successful CAS. In this case, the first thread that exits the parallel section

will be successful, and the *Access* of the sequence will be a *Read*. Second, the contention of a success period is high when at any time during the success period, there exists a thread that is executing a *CAS*. In other words, at the end of each *CAS*, there is at least one thread that is waiting for the cache line to operate a *CAS* on it. This implies that the first access of the success period is a *CAS* and occurs immediately after the preceding successful *CAS*: the slack time is null. Third, the mid-contention mode takes place when $P_{rl} > 0$, while at the same time, there are not enough requesting threads to fill the whole success period with *CAS*'s (which implies a non-null slack time). Since these requesting threads have synchronized in the previous success period, *CAS*'s do not collide in the current success period, and because of that, the expansion is null.

4 Average-based Approach

We propose in this section our coarse-grained analysis to predict the performance of lock-free data structures. Our approach utilizes fundamental queuing theory techniques, describing the average behavior of the algorithm. In turn, we need only a minimal knowledge about the algorithm: the mean execution time values *cw* and *pw*. As explained in Section 3.3.1, the system runs in one of the two possible modes: either contended or uncontended.

4.1 Contended System

We first consider a system that is contended. When the system is contended, we use Little's law to obtain, at a random time, the expectation of the success period, which is the interval of time between the last and the next successful *CAS*'s (see Figure 2).

The stable system that we observe is the parallel section: threads are entering it (after exiting a successful retry loop) at an average rate, stay inside, then leave (while entering a new retry loop). The average number of threads inside the parallel section is $\overline{P_{ps}} = P - \overline{P_{rl}}$, each thread stays for an average duration of *pw*, and in average, one thread is exiting the retry loop every success period $\overline{sp}\left(\overline{P_{rl}}\right)$, by definition of the success period.

According to Little's law [14], we have:

$$\overline{P_{ps}} = pw \times 1/\overline{sp}\left(\overline{P_{rl}}\right), \text{ i.e. } \overline{sp}\left(\overline{P_{rl}}\right) = pw/(P - \overline{P_{rl}}). \tag{1}$$

We decompose a success period into two parts: slack time and completion time (as explained in Section 3.2). We express the expectation of the success period time as

$$\overline{sp}\left(\overline{P_{rl}}\right) = \overline{st}\left(\overline{P_{rl}}\right) + \overline{ct}\left(\overline{P_{rl}}\right). \tag{2}$$

When the data structure is contended, a thread is likely to be successful after some failed retries. Therefore a thread that is successful was already in the retry loop when the previous successful *CAS* occurred. The time before a thread starts its *Access* is then the time before a thread finishes its current critical work since there is a thread currently executing a *CAS*.

4.1.1 Expected Completion Time

Since the data structure is contended, numerous threads are inside the retry loop, and, due to hardware conflicts, a retry can experience expansion: the more threads inside the retry loop, the longer time between a *CAS* request and the actual execution of this *CAS*. The expectation of the completion time can be written as:

$$\overline{ct}\left(\overline{P_{rl}}\right) = cc + cw + \overline{e}\left(\overline{P_{rl}}\right) + cc, \tag{3}$$

where $\overline{e}\left(\overline{P_{rl}}\right)$ is the expectation of expansion when there are $\overline{P_{rl}}$ threads inside the retry loop, in expectation. This expansion can be computed in the same way as in [3], through the following differential equation:

$$\begin{cases} \overline{e}'\left(\overline{P_{rl}}\right) & = & cc \times \dfrac{cc/2 + \overline{e}\left(\overline{P_{rl}}\right)}{cc + cw + cc + \overline{e}\left(\overline{P_{rl}}\right)}, \\ \overline{e}\left(1\right) & = & 0 \end{cases}$$

by assuming that the expansion starts as soon as strictly more than 1 thread are in the retry loop, in expectation.

4.1.2 Expected Slack Time

Concerning the slack time, we consider that, at any time, the threads that are running the retry loop have the same probability to be anywhere in their current retry. However, when a thread is currently executing a CAS, the other threads cannot execute as well a CAS. The other threads are thus in their critical work or expansion. For every thread, the time before accessing the data structure is then uniformly distributed between 0 and $cw + \overline{e}\left(\overline{P_{rl}}\right)$. Using a well-known formula on the expectation of the minimum of uniformly distributed random variables, we show in [4] that:

$$\overline{st}\left(\overline{P_{rl}}\right) = \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right)/(\overline{P_{rl}} + 1). \tag{4}$$

4.1.3 Expected Success Period

We just have to combine Equations 2, 3, and 4 to obtain the general expression of the expected success period under contention: $\overline{sp}\left(\overline{P_{rl}}\right) = \left(1 + 1/(\overline{P_{rl}} + 1)\right)\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc$, which leads, according to Equation 1, to

$$\frac{1}{pw} \times \left(\frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc\right) = \frac{1}{P - \overline{P_{rl}}}. \tag{5}$$

4.2 Non-contended System

When the system is not contended, logical conflicts are not likely to happen, hence each thread succeeds in its retry loop at its first *retry*. *A fortiori*, no hardware conflict occurs. Each thread still performs one success every work loop, and the success period is given by $\overline{sp}\left(\overline{P_{rl}}\right) = (pw + rc + cw + cc)/P$. Moreover, a thread spends in average pw units of time in the retry loop within each work loop. As this holds for every thread, we deduce $P - \overline{P_{rl}} = \overline{P_{ps}} = pw/(pw + rc + cw + cc) \times P$. Combining the two previous equations, we obtain

$$\frac{\overline{sp}\left(\overline{P_{rl}}\right)}{pw} = \frac{1}{P - \overline{P_{rl}}}, \text{ where } \overline{sp}\left(\overline{P_{rl}}\right) = \frac{rc + cw + cc}{\overline{P_{rl}}}. \tag{6}$$

4.3 Unified Solving

We have to decide whenever the data structure is under contention or not, and to find the corresponding solution. Concerning the frontier between contended and non-contended system, we can remark that Equations 5 and 6 are equivalent if and only if

$$\frac{rc + cw + cc}{\overline{P_{rl}}} = \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1}\left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) + 2cc, \tag{7}$$

which leads to Lemma 1.

▶ **Lemma 1.** *The system switches from being non-contended to being contended at* $\overline{P_{rl}} = P_{rl}^{(0)}$, *where*

$$P_{rl}^{(0)} = \frac{cc + cw - rc}{2(cw + 2cc)} \left(\sqrt{1 + \frac{4(rc + cw + cc)(cw + 2cc)}{(cc + cw - rc)^2}} - 1 \right).$$

Proof. The three following properties, proved in [4], demonstrate the lemma:
(i) $P_{rl}^{(0)}$ is the unique positive solution of Equation 7 if the expansion is set to 0,
(ii) $P_{rl}^{(0)} \leq 1$, and
(iii) there is no solution of Equation 7 with a non-null expansion. ◀

Thanks to Lemma 1, we can unify the success period as:

$$\overline{sp}\left(\overline{P_{rl}}\right) = \begin{cases} \left(rc + cw + cc\right) / \overline{P_{rl}} & \text{if } \overline{P_{rl}} \leq P_{rl}^{(0)} \\ \left(cw + \overline{e}\left(\overline{P_{rl}}\right)\right) \times \frac{\overline{P_{rl}+2}}{\overline{P_{rl}+1}} + 2cc & \text{otherwise.} \end{cases}$$

The unified success period obeys to the following equation

$$\overline{sp}\left(\overline{P_{rl}}\right) = \frac{pw}{P - \overline{P_{rl}}}. \tag{8}$$

We show in the following theorem how to compute the throughput estimate; the proof, presented in [4], manipulates equations in order to be able to use the fixed-point Knaster-Tarski theorem.

▶ **Theorem 2.** *The throughput can be obtained iteratively through a fixed-point search, as* $T = \left(\overline{sp}\left(\lim_{n\to+\infty} u_n\right)\right)^{-1}$, *where*

$$\begin{cases} u_0 = \frac{rc+cw+cc}{pw+rc+cw+cc} \times P \\ u_{n+1} = \frac{u_n \overline{sp}(u_n)}{pw+u_n \overline{sp}(u_n)} \times P \qquad \textit{for all } n \geq 0. \end{cases}$$

5 Constructive Approach

In this section, we instantiate the probability distribution of the parallel work with an exponential distribution. We have therefore a better knowledge of the behavior of the algorithm, particularly in medium contention cases, which allows us to follow a fine-grained approach that studies individually each successful operation together with every *CAS* occurrence. We provide an elegant and efficient solution that relies on a Markov chain analysis.

5.1 Process

We have seen in Section 3.3.2 that we split the contention domain into three modes: no contention, medium contention or high contention. We start from a configuration with a given number of threads P_{rl} after a successful *CAS*, and describe what will happen until the next successful *CAS*: what will be the mode of the next success period, and more precisely, which will be the number of threads at the beginning of the next success period.

As a basis, we consider the execution that would occur without any other thread exiting the parallel section (then entering the retry loop); we call this execution the *internal execution*. This execution follows the success period pattern described in Figure 2 (with an infinite slack time if the system is not contended). On top of this basic success period, we inject

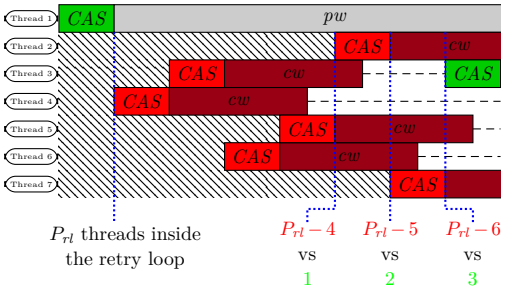


Figure 3 Highly-contended execution.

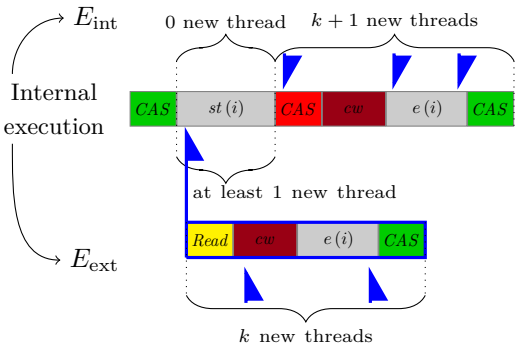


Figure 4 Possible executions.

the threads that can exit the parallel section, which has a double impact. On the one hand, they increase the number of threads inside the retry loop for the next success period. On the other hand, if the first thread that exits the parallel section starts its retry during the slack time of the success period of the internal execution, then this thread will succeed its *Access*, which is a *Read*, and will shrink the actual slack time of the current success period.

According to the distribution probability of the arrival of the new threads, we can compute the probability for the next success period to start with any number of threads. The expression of this stochastic sequence of success periods in terms of Markov chains results in the throughput estimate.

5.2 Expansion

The expansion, as before, represents the additional time in the execution time of a retry, due to the serialization of atomic primitives. However, in contrary to Section 4.1.1, we compute here this additional time in the current success period, according to the number of threads P_{rl} inside the retry loop at the beginning of the success period. The expansion only appears when the success period is highly contended, *i.e.* when we can find a continuous sequence of *CAS*'s all through the success period.

The expansion is highly correlated with the way the cache coherence protocol handles the exchange of cache lines between threads. We rely on the experiments of the research report associated with [2], which show that if several threads request for the same cache line in order to operate a *CAS*, while another thread is currently executing a *CAS*, they all have an equal probability to obtain the cache line when the current *CAS* is over.

We draw an illustrative example in Figure 3. The green *CAS*'s are successful while the red *CAS*'s fail. To lighten the picture, we hide what happened for the threads before they experience a failed *CAS*. The horizontal dash lines represent the time where a thread wants to access the data in order to operate a *CAS* but has to wait because another thread owns the data in exclusive mode. We can observe in this example that the first thread that accesses the data structure is not the thread whose operation returns.

We are given that P_{rl} threads are inside the retry loop at the end of the previous successful *CAS*, and we only consider those threads. When such a thread executes a *CAS* for the first time, this *CAS* is unsuccessful. The thread was in the retry loop when the successful *CAS* has been executed, so it has read a value that is not up-to-date anymore. However, this failed *CAS* will bring the current version of the value (to compare-and-swap) to the thread, a value that will be up-to-date until a successful *CAS* occurs.

So we have firstly a sequence of failed *CAS*'s until the first thread that operated its *CAS* within the current success period finishes its critical work. At this point, there exists a thread that is executing a *CAS*. When this *CAS* is finished, some threads compete to obtain the cache line. We have two bags of competing threads: in the first bag, the thread that just ended its critical work is alone, while in the second bag, there are all the threads that were in the retry loop at the beginning of the success period, and did not operate a *CAS* yet. The other, non-competing, threads are running their critical work and do not yet want to access the data.

As described before, every thread has the same probability to become the next owner of the cache line. If a thread from the first bag is drawn, then the *CAS* will be successful and the success period ends. Otherwise, the *CAS* is a failure, and we iterate at the end of this failed *CAS*. However, the thread that just failed its *CAS* is now executing its critical work, and does not request for a new *CAS* until this work has been done, thus it is not anymore in the second bag. In addition, the thread that had executed its *CAS* after the thread of the first bag is now back from its critical work and falls into the first bag. The process iterates until a thread is drawn from the first bag.

As a remark, note that we do not consider threads that are not in the retry loop at the beginning of the success period since even if they come back from the parallel section during the success period, their *Read* will be delayed and their *CAS* is likely to occur after the end of the success period.

Theorem 3, proved in [4], gives the explicit formula for the expansion.

▶ **Theorem 3.** *The expected time between the end of the critical work of the first thread that operates a CAS in the success period and the beginning of a successful CAS is given by:*

$$e\left(P_{rl}\right) = \lceil cw/cc \rceil cc - cw + \sum_{i=1}^{P_{com}} \frac{i(i-1)}{\left(P_{com}\right)^i} \frac{\left(P_{com}-1\right)!}{\left(P_{com}-i\right)!} \times cc, \quad \text{where } P_{com} = P_{rl} - \lceil cw/cc \rceil + 1.$$

5.3 Formalization

The parallel work follows an exponential distribution, whose mean is pw. More precisely, if a thread starts a parallel section at the instant t_1, the probability distribution of the execution time of the parallel section is $t \mapsto \lambda e^{-\lambda(t-t_1)} \mathbb{1}_{[t_1,+\infty[}(t)$, where $\lambda = 1/pw$. This probability distribution is memoryless, which implies that the threads that are executing their parallel section cannot be differentiated: at a given instant, the probability distribution of the remaining execution time is the same for all threads in the parallel section, regardless of when the parallel section began. For all threads, it is defined by: $t \mapsto \lambda \exp\left(-\lambda t\right)$, where $\lambda = 1/pw$.

For the behavior in the retry loop, we rely on the same approximation as in the previous section, *i.e.* when a successful thread exits its retry loop, the remaining execution time of the retry of every other thread that is still in the retry loop is uniformly distributed between 0 and the execution time of a whole retry. We have seen that the expectation of this remaining time is the size of the execution time of a retry divided by the number of threads inside the retry loop plus one. Here, we assume that a thread will start a retry at this time. This implies another kind of memoryless property: the behavior of a thread that is in the retry loop does not depend on the moment that it entered its retry loop.

To tackle the problem of estimating the throughput of such a system, we use an approach based on Markov chains. We study the behavior of the system over time, step by step: a state of the Markov chain represents the state of the system when the current success

period began (*i.e.* just after a successful *CAS*) and (thus) the system changes state at the end of every successful *CAS*. According to the current state, we are able to compute the probability to reach any other state at the beginning of the next success period. In addition, the two memoryless properties render the description of a state easy to achieve: the number of threads inside the retry loop when the current success begins, indeed fully characterizes the system.

We recall that P_{rl} is the number of threads inside the retry loop when the success period begins. The Markov chain is strongly related with P_{rl}, since it is composed of P states $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{P-1}$, where, for all $i \in [\![0, P-1]\!]$, the success period is in state $\mathcal{S}_i$ iff $P_{rl} = i$. For all $(i,j) \in [\![0, P-1]\!]^2$, $\mathbb{P}(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the probability that a success characterized by $\mathcal{S}_j$ follows a success in state $\mathcal{S}_i$. $st(\mathcal{S}_i \to \mathcal{S}_j)$ denotes the slack time that passed while the system has gone from state $\mathcal{S}_i$ to state $\mathcal{S}_j$. This slack time can be expressed based on the slack time $st(i)$ of the internal execution, *i.e.* the execution that involves only the i threads of the retry loop and ignores the other threads (see Section 5.1). In the same way, we denote by $ct(i)$ the completion time of the internal execution, hence $ct(i) = cc + cw + e(i) + cc$.

We have seen that the level of contention (mode) is determined by P_{rl}, hence the interval $[\![0, P-1]\!]$ can be partitioned into $[\![0, P-1]\!] = \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{hi}}$, where the partitions correspond to the different contention levels. So, by definition, $\mathcal{I}_{\text{noc}} = \{0\}$, and for all $i \in \mathcal{I}_{\text{noc}} \cup \mathcal{I}_{\text{mid}}$, $e(i) = 0$ (see Section 3.3.2).

The success period is highly-contended, *i.e.* we have a continuous sequence of *CAS*'s in the success period, if the sum of the execution time of all the *CAS*'s that need to be operated exceeds the critical work. Hence $\mathcal{I}_{\text{hi}} = [\![i_{\text{hi}}, P-1]\!]$, where $i_{\text{hi}} = \min\{i \in [\![1, P-1]\!] \mid i \times cc > cw\}$. In addition, as the sequence of *CAS*'s is continuous when the contention is high, the slack time is null when the success period is highly contended, *i.e.*, for all $i \in \mathcal{I}_{\text{hi}}$, $st(i) = 0$, and *a fortiori*, $st(\mathcal{S}_i \to \mathcal{S}_\star) = 0$.

Otherwise, the success period is in medium contention, hence $\mathcal{I}_{\text{mid}} = [\![1, i_{\text{hi}} - 1]\!]$. Moreover, if $i \in \mathcal{I}_{\text{mid}}$, $st(i) > 0$, and $e(i) = 0$, because the *CAS*'s synchronized during the previous success period and will not collide any more in the current success period.

Everything is now in place to be able to obtain the stationary distribution of the Markov chain, and in turn to compute the throughput and the failure rate estimates. The reasoning that leads to the computation of the probability of going from state $\mathcal{S}_i$ to state $\mathcal{S}_{i+k}$ can be roughly summarized by Figure 4, where we start from an internal execution with i threads inside the retry loop and the blue arrows represent the threads that exit the parallel section. Two non-overlapping events can then potentially occur: either (event E_{ext}) the first thread exiting the parallel section starts within $[0, st(i)[$, *i.e.* in the slack time of the internal execution, or (event E_{int}) the first thread entering the retry loop starts after $t = st(i)$. The details can be found in [4].

6 Experiments

To validate our analysis results, we use two main types of lock-free algorithms. In the first place, we consider a set of basic algorithms where operations can be completed with a single successful *CAS*. This set of algorithms includes:

- **(i)** synthetic designs, that cover the design space of possible lock-free data structures;
- **(ii)** several fundamental designs of data structure operations such as lock-free stacks [21] (Pop, Push), queues [17] (Dequeue), counters [8] (Increment, Decrement).

As a second step, we consider more advanced lock-free operations that involve helping mechanisms, and show how to use our analysis in this context. Finally, in order to highlight

the benefits of the analysis framework, we show how it can be applied to

(i) determine a beneficial back-off strategy and

(ii) optimize the memory management scheme used by a data structure, in the context of an application.

We also give insights about the strengths of our two approaches. The constructive approach exhibits better predictions due to the tight estimation of the failing retries. On the other hand, the average-based approach is applicable to a broader spectrum of algorithmic designs as it leaves room to abstract complicated algorithmic designs.

6.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets equipped with Intel Xeon E5-2687W v2 CPUs. In a socket, the ring interconnect provides L3 cache accesses and core-to-core communication. Threads are pinned to a single socket to minimize non-uniformity in *Read* and *CAS* latencies. The methodology in [7] is used to measure the *CAS* and *Read* latencies, while the parallel work is implemented by a for-loop of *Pause* instructions. We show the experimental results with 8 threads.

In all figures, the y-axis shows both the throughput, *i.e.* number of operations completed per second, and the ratio of failing to successful retries (multiplied by 10^6, for readability), while the mean of the exponentially distributed parallel work pw is represented on the x-axis. The number of failures per success in the average-based approach is computed as $\overline{P_{rl}} - 1$ and in the constructive approach by stochastically counting the failing *CAS*'s inside a success period (see [4]).

We have also added a straightforward upper bound as a baseline approach, defined as the minimum of $1/(rc + cw + cc)$ (two successful retries cannot overlap) and $P/(pw + rc + cw + cc)$ (a thread can succeed only once in each work loop).

6.2 Basic Data Structures

Firstly, we consider lock-free operations that can be completed with a single successful *CAS*. We provide predictions, on the one hand, on a set of synthetic tests that have been constructed to abstract different possible design patterns of lock-free data structures (value of cw) and different application contexts (value of pw), and, on the other hand, on the well-known Treiber stack. The results, that show the satisfactory quality of the prediction, are depicted in [4].

6.3 Towards Advanced Data Structure Designs

Advanced lock-free operations generally require multiple pointer updates that cannot be done with a single *CAS*. One way to design such operations, in a lock-free manner, is to use helping mechanisms: an inconsistency will be fixed eventually by some thread. Here we consider two data structures that apply immediate helping, the queue from [17] and the deque designed in [15]. In the queue experiment (Figure 5), we run the Enqueue operation on the queue with and without memory management; in the deque experiment, each thread is dedicated to an end of the deque (equally distributed), while we vary the proportion of push operations (colors in Figure 6). More details about the implementations and the throughput estimate obtained through a slight modification of the average-based approach can be found in [4].

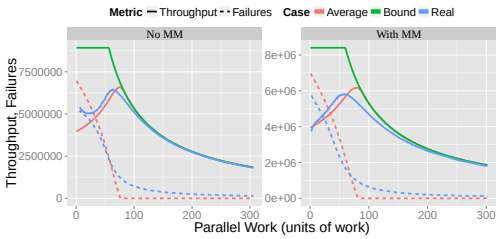


Figure 5 Enqueue on MS queue.

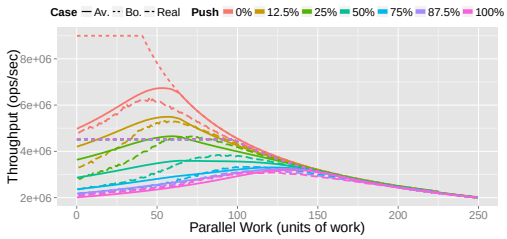


Figure 6 Operations on deque.

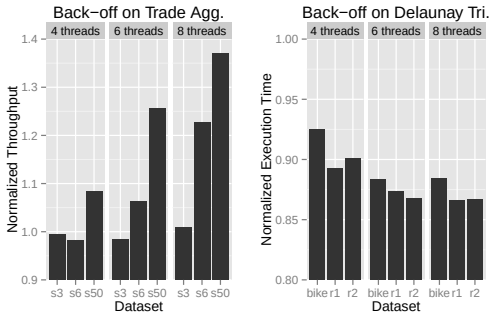


Figure 7 Performance impact of our back-off tunings.

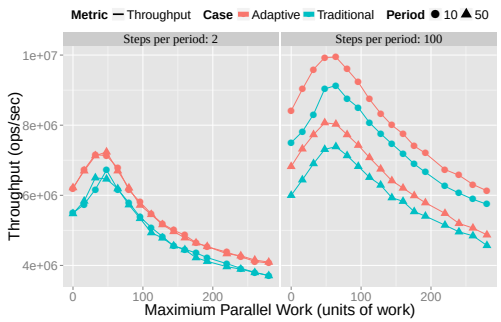


Figure 8 Adaptive MM with varying mean pw.

6.4 Applications

6.4.1 Back-off Optimizations

When the parallel work is known, we can deduce from our analysis a simple and efficient back-off strategy: as we are able to estimate the value for which the throughput is maximum, we just have to back-off for the time difference between the peak pw and the actual pw. In [4], we compare this back-off strategy against widely known strategies, namely exponential and linear, on a synthetic workload. In Figure 7, we apply our constant back-off on a Delaunay triangulation application [9], provided with several workloads. The application uses a stack in two phases, whose first phase pushes elements on top of the stack without delay. We are able to estimate a corresponding back-off time, and we plot the results by normalizing the execution time of our back-offed implementation with the execution time of the initial implementation.

A measure or an estimate of pw is not always available (and could change over time, see next section), therefore we propose also an adaptive strategy: we incorporate in the data structure a monitoring routine that tracks the number of failed retries, employing a sliding window. As our analysis computes an estimate of the number of failed retries as a function of pw, we are able to estimate the current pw, and hence the corresponding back-off time like previously.

We test our adaptive back-off mechanism on a workload originated from [19], where global operators of exchanges for financial markets gather data of trades with a microsecond accuracy. We assume that the data comes from several streams, each of them being associated with a thread. All threads enqueue the elements that they receive in a concurrent queue, so that they can be later aggregated. We extract from the original data a trade stream distribution that

we use to generate similar streams that reach the same thread; varying the number of streams to the same thread leads to different workloads. The results, represented as the normalized throughput (compared to the initial throughput) of trades that are enqueued when the adaptive back-off is used, are plotted in Figure 7. For any number of threads, the queue is not contended on workload s3, hence our improvement is either small or slightly negative. On the contrary, the workload s50 contends the queue and we achieve very significant improvement.

6.4.2 Memory Management Optimization

Memory Management (MM) is an inseparable part of dynamic concurrent data structures. In contrary to lock-based implementations, a node that has been *removed* from a lock-free data structure can still be accessed by other threads, *e.g.* if they have been delayed. Collective decisions are thus required in order to *reclaim* a node in a safe manner. A well-known solution to deal with this problem is the hazard pointers technique [16]. In an implementation of such design each thread lists the nodes that it accesses and the nodes that it has removed. When the number of nodes it has removed reaches a threshold, it reclaims its listed removed nodes that are not listed as accessed by any thread.

The main goal of our adaptive MM scheme is to distribute this extra-work in a way that the loss in performance is largely leveraged, knowing that additional work can be an advantage under high-contention (see previous section). The optimization is based on two main modifications. First, we divide the reclamation phase of the traditional MM scheme into quanta (equally-sized chunks), whose finer granularity allows for accurate back-off times. Second, we track continuously the contention level in the same way as our adaptive back-off. See [4].

We emulate the behavior of many scientific applications, that are built upon a pattern of alternating phases, that are communication-intensive (synchronization phase) or computation-intensive. Here we assume a synchronization ensured through a shared data structure, hence the communication-intensive phases correspond to a high access rate to the data structure, while the data structure is accessed at a low rate during a computation-intensive phase. The parallel work still follows an exponential distribution of mean pw, but pw varies in a sinusoidal manner with time. To study also the impact of the continuity of the change in pw, pw is set as a step approximation of a sine function. Thus, two additional parameters rule the experiment: the period of the oscillating function represents the length of the phases, and the number of steps within a period depicts how continuous the phase changes are.

In Figure 8, we compare our approach with the traditional implementation for different periods of the sine function, on the Dequeue of the Michael-Scott queue [17]. The adaptive MM, that relies on the analysis presented in this paper, outperforms the traditional MM because it provides an advantage both under low contention due to the costless (since delayed) invocation of the MM and under high contention due to the back-off effect.

7 Conclusion

In this paper we have presented two analyses for calculating the performance of lock-free data structures in dynamic environments. The first analysis has its roots in queuing theory, and gives the flexibility to cover a large spectrum of configurations. The second analysis makes use of Markov chains to exhibit a stochastic execution; it gives better results, but it is restricted to simpler data structures and exponentially distributed parallel work. We have shown how to use our results to tune applications using lock-free codes. These tuning methods include:

(i) the calculation of simple and efficient back-off strategies whose applicability is illustrated in application contexts;
(ii) a new adaptive memory management mechanism that acclimates to a changing environment.

The main differences between the data structures of this paper and linked lists, skip lists and trees occur when the size of the data structure grows. With large sizes, the performance is dominated by the traversal cost that is ruled by the cache parameters. The reduction in the size of the data structure decreases the traversal cost which in turn increases the probability of encountering an on-going *CAS* operation that delays the threads which traverse the link. The expansion, which can additionally be supported unfavorably by helping mechanisms, appears then as the main performance degrading factor. While the analysis becomes easier for high degrees of parallelism (large data structure size), being able to describe the behavior of lock-free data structures as the degree of parallelism changes constitutes the main challenge of our future work.

References

1 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002. doi:10.1007/s004460100060.
2 Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In *STOC*, pages 714–723. ACM, Jun 2014.
3 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the performance of lock-free data structures: A conflict-based model. In *DISC*, pages 341–355, 2015.
4 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. How lock-free data structures perform in dynamic environments: Models and analyses. Technical Report 2016:10, Chalmers University of Technology, Nov 2016. URL: http://arxiv.org/abs/1611.05793.
5 Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *JACM*, 50(4):444–468, 2003. doi:10.1145/792538.792541.
6 Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC*, pages 122–136, 2005. doi:10.1007/11561927_11.
7 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48. ACM, Nov 2013.
8 Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *SPAA*, pages 43–52. ACM, Jul 2013.
9 Tanmay Gangwani, Adam Morrison, and Josep Torrellas. CASPAR: breaking serialization in lock-free multicore synchronization. In *ASPLOS*, pages 789–804, 2016.
10 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Par. Distr. Computing*, 70(1):1–12, 2010.
11 Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991.
12 Intel. Threading building blocks framework. Accessed: 2016-01-20. URL: https://www.threadingbuildingblocks.org/.
13 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PoDC*, pages 151–160, 1994.
14 John D. C. Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.
15 Maged M. Michael. Cas-based lock-free algorithm for shared deques. In *Euro-Par*, pages 651–660, 2003. doi:10.1007/978-3-540-45209-6_92.
16 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(8), Aug 2004.

17 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PoDC*, pages 267–275. ACM, May 1996.

18 Microsoft. NET Framework. Accessed: 2016-01-20. URL: `http://www.microsoft.com/net`.

19 NYSE. Daily trades from 2015-08-05. Accessed: 2016-05-05. URL: `http://www.nyxdata.com/Data-Products/Daily-TAQ#155`.

20 Oracle. Java concurrency package. Accessed: 2016-01-20. URL: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`.

21 R. Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

22 J. D. Valois. Implementing Lock-Free Queues. In *ICPADS*, pages 64–69, Dec 1994.

Non-Determinism in Byzantine Fault-Tolerant Replication*

Christian Cachin[1], Simon Schubert[2], and Marko Vukolić[3]

1 **IBM Research, Zürich, Switzerland**
`cca@zurich.ibm.com`
2 **IBM Research, Zürich, Switzerland**
`sis@zurich.ibm.com`
3 **IBM Research, Zürich, Switzerland**
`mvu@zurich.ibm.com`

Abstract

Service replication distributes an application over many processes for tolerating faults, attacks, and misbehavior among a subset of the processes. With the recent interest in blockchain technologies, distributed execution of one logical application has become a prominent topic. The established state-machine replication paradigm inherently requires the application to be deterministic. This paper distinguishes three models for dealing with non-determinism in replicated services, where some processes are subject to faults and arbitrary behavior (so-called Byzantine faults): first, the modular case that does not require any changes to the potentially non-deterministic application (and neither access to its internal data); second, master-slave solutions, where ties are broken by a leader and the other processes validate the choices of the leader; and finally, applications that use cryptography and secret keys. Cryptographic operations and secrets must be treated specially because they require strong randomness to satisfy their goals.

The paper also introduces two new protocols. First, Protocol *Sieve* uses the modular approach and filters out non-deterministic operations in an application. It ensures that all correct processes produce the same outputs and that their internal states do not diverge. A second protocol, called *Mastercrypt*, implements cryptographically secure randomness generation with a verifiable random function and is appropriate for most situations in which cryptographic secrets are involved. All protocols are described in a generic way and do not assume a particular implementation of the underlying consensus primitive.

1998 ACM Subject Classification C.2.4 Distributed Systems, D.1.3 Concurrent Programming

Keywords and phrases Blockchain, atomic broadcast, consensus, distributed cryptography, verifiable random functions

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.24

1 Introduction

State-machine replication is an established way to enhance the resilience of a client-server application [33]. It works by executing the service on multiple independent components that will not exhibit correlated failures. We consider the approach of *Byzantine fault-tolerance (BFT)*, where a group of *processes* connected only by an unreliable network executes an

* This work was supported in part by the European Union's Horizon 2020 Framework Programme under grant agreement number 643964 (SUPERCLOUD) and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

application [31]. The processes use a protocol for *consensus* or *atomic broadcast* to agree on a common sequence of operations to execute. If all processes start from the same initial state, if all operations that modify the state are *deterministic*, and if all processes execute the same sequence of operations, then the states of the correct processes will remain the same. (This is also called *active* replication [13].) A client executes an operation on the service by sending the operation to all processes; it obtains the correct outcome based on comparing the responses that it receives, for example, by a relative majority among the answers or from a sufficiently large set of equal responses. Tolerating *Byzantine faults* means that the clients obtain correct outputs as long as a qualified majority of the processes is correct, even if the faulty processes behave in arbitrary and adversarial ways.

Traditionally state-machine replication requires the application to be deterministic. But many applications contain implicit or explicit non-determinism: in multi-threaded applications, the scheduler may influence the execution, input/output operations might yield different results across the processes, probabilistic algorithms may access a random-number generator, and some cryptographic operations are inherently not deterministic.

Recently BFT replication has gained prominence because it may implement distributed consensus for building *blockchains* [1, 15, 35, 5]. A blockchain provides a distributed, append-only ledger with cryptographic verifiability and is governed by decentralized control. It can be used to record events, trades, or transactions immutably and permanently and forms the basis for cryptocurrencies, such as Bitcoin or Ripple, or for running "smart contracts," as in Ethereum. With the focus on active replication, this work aims at *permissioned* blockchains, which run among known entities [34]. In contrast, *permissionless* blockchains (including Ethereum) do not rely on identities and use other approaches for reaching consensus, such as proof-of-work protocols. For practical use of blockchains, ensuring deterministic operations is crucial since even the smallest divergence among the outputs of different participants lets the blockchain diverge (or "fork").

This work presents a *general treatment* of non-determinism in the context of BFT replication and introduces a distinction among different models to tackle the problem of non-determinism. For example, applications involving cryptography and secret encryption keys should be treated differently from those that access randomness for other goals. We also distinguish whether the replication mechanism has access to the application's source code and may modify it.

We also introduce two novel protocols. The first, called *Sieve*, replicates non-deterministic programs using in a *modular* way, where we treat the application as a black box and cannot change it. We target workloads that are usually deterministic, but which may occasionally yield diverging outputs. The protocol initially executes all operations speculatively and then compares the outputs across the processes. If the protocol detects a minor divergence among a small number of processes, then we *sieve out the diverging values*; if a divergence among too many processes occurs, we *sieve out the operation* from sequence. Furthermore, the protocol can use *any* underlying consensus primitive to agree on an ordering. The second new protocol, *Mastercrypt*, provides master-slave replication with cryptographic security from verifiable random functions. It addresses situations that require strong, cryptographically secure randomness, but where the faulty processes may leak their secrets.

1.1 Contributions

We introduce three different models and discuss corresponding protocols for replicating non-deterministic applications.

Modular: When the application itself is fixed and cannot be changed, then we need *modular* replicated execution. In practice this is often the case. We distinguish two approaches for integrating a consensus protocol for ordering operations with the replicated execution of operations. One can either use *order-then-execute*, where the operations are ordered first, executed independently, and the results are communicated to the other processes through atomic broadcast. This involves only deterministic steps and can be viewed as "agreement on the input." Alternatively, with *execute-then-order*, the processes execute all operations speculatively first and then "agree on the output" (of the operation). In this case operations with diverging results may have to be rolled back.

We introduce Protocol *Sieve* that uses speculative execution and follows the *execute-then-order* approach. As described before, *Sieve* is intended for applications with occasional non-determinism. It represents the first modular solution to replicating non-deterministic applications in a BFT system.

Master-slave: In the *master-slave* model, one process is designated as the master or "leader," makes all non-deterministic choices that come up, and imposes these on the others which act as slaves or "followers." Because a faulty (Byzantine) master may misbehave, the slaves must be able to validate the selections of the master before the operation can be executed as determined by the master. The master-slave model is related to passive replication; it works for most applications including probabilistic algorithms, but cannot be applied directly for cryptographic operations. As a further complication, this model requires that the developer has access to the internals of the application and can modify it.

For the master-slave model we give a detailed description of the well-known replication protocol, which has been used in earlier systems.

Cryptographically secure: Traditionally, randomized applications can be made deterministic by deriving pseudorandom bits from a secret seed, which is initially chosen truly randomly. Outsiders, such as clients of the application, cannot distinguish this from an application that uses true randomness. This approach does not work for BFT replication, where faulty processes might expose and leak the seed. To solve this problem, we introduce a novel protocol for master-slave replication with cryptographic randomness, abbreviated *Mastercrypt*. It lets the master select random bits with a *verifiable random function*. The protocol is aimed at applications that need strong, cryptographically secure randomness; however it does not protect against a faulty master that leaks the secret. We also review the established approach of threshold (public-key) cryptography, where private keys are secret-shared among the processes and cryptographic operations are distributed in a fault-tolerant way over the whole group.

The modular Protocol *Sieve* has been developed for running potentially non-deterministic smart contracts as applications on top of a permissioned blockchain platform, built using BFT replication. An implementation has been made available as open source in "Hyperledger fabric" (`https://github.com/hyperledger/fabric`), which is part of the Linux Foundation's Hyperledger Project. As of November 2016, the project has decided to adopt a different architecture (`https://github.com/hyperledger/fabric/blob/master/docs/`); the platform has been redesigned to use a master-slave approach for addressing non-deterministic execution.

1.2 Related work

The problem of ensuring deterministic operations for replicated services is well-known. When considering only crash faults, many authors have investigated methods for making services

deterministic, especially for multi-threaded, high-performance services [2]. Practical systems routinely solve this problem today using master-slave replication, where the master removes the ambiguity and sends deterministic updates to the slaves. In recent research on this topic, for instance, Kapitza et al. [24] present an optimistic solution for making multithreaded applications deterministic. Their solution requires a predictor for non-deterministic choices and may invoke additional communication via the consensus module.

In the BFT model, most works consider only sequential execution of deterministic commands, including PBFT [10] and UpRight [14]. BASE [11] and CBASE [26] address Byzantine faults and adopt the master-slave model for handling non-determinism, focusing on being generic (BASE) and on achieving high throughput (CBASE), respectively. These systems involve changes to the application code and sometimes also need preprocessing steps for operations.

Fault-tolerant execution on multi-core servers poses a new challenge, even for deterministic applications, because thread-level parallelism may introduce unpredictable differences between processes. Eve [25] heuristically identifies groups of non-interfering operations and executes each group in parallel. Afterwards it compares the outputs, may roll back operations that lead to diverging states, or could transfer an agreed-on result state to diverging processes. Eve resembles Protocol *Sieve* in this sense, but lacks modularity.

For the same domain of scalable services running on multi-cores, Rex [21] uses the master-slave model, where the master executes the operations first and records its non-deterministic choices. The slaves replay these operations and use a consensus primitive to agree on a consistent outcome. Rex only tolerates crashes, but does not address the BFT model.

Fault-tolerant replication involving cryptographic secrets and distributed cryptography has been pioneered by Reiter and Birman [32]. Many other works followed, especially protocols using threshold cryptography; an early overview of solutions in this space was given by Cachin [4].

In current work Duan and Zhang [18] discuss how the master-slave approach can handle randomized operations in BFT replication, where execution is separated from agreement in order to protect the privacy of the data and computation.

1.3 Organization

The remainder of this paper starts with Section 2, containing background information and formal definitions of broadcast, replication, and atomic broadcast (i.e., consensus). The following sections contain the discussion and protocols for the three models: the modular solution (Section 3), the master-slave protocol (Section 4), and replication methods for applications demanding cryptographic security (Section 5). Due to lack of space in this extended abstract, many details, definitions, protocol optimizations, and discussions are only available in the full version [9].

2 Definitions

2.1 System model

We consider a distributed system of *processes* that communicate with each other and provide a common *service* in a fault-tolerant way. Using the paradigm of service replication [33], requests to the service are broadcast among the processes, such that the processes execute all requests in the same order. The clients accessing the service are not modeled here. We denote the set of processes by $\mathcal{P}$ and let $n = |\mathcal{P}|$. A process may be *faulty*, by crashing

or by exhibiting *Byzantine faults*; the latter means they may deviate arbitrarily from their specification. Non-faulty processes are called *correct*. Up to f processes may be faulty and we assume that $n > 3f$. The setup is also called a *Byzantine fault-tolerant (BFT) service replication system* or simply a *BFT system*.

We present protocols in a modular way using an event-based notation [6]. A process is specified through its interface, consumes input events, and generates output events. Every two processes can *send* messages to each other using an authenticated point-to-point communication primitive. When a message arrives, the receiver learns also which process has sent the message. The primitive guarantees *message integrity*, i.e., when a message m is received by a correct process with indicated sender p_s, and p_s is correct, then p_s previously sent m.

The system is *partially synchronous* [19] in the sense that there is no a priori bound on message delays and the processes have no synchronized clocks, as in an asynchronous system. However, there is a time (not known to the processes) after which the system is *stable* in the sense that message delays and processing times are bounded. In other words, the system is *eventually synchronous*. This model represents a broadly accepted network model and covers a wide range of real-world situations.

2.2 Broadcast and state-machine replication

Atomic broadcast. Suppose n processes participate in a broadcast primitive. Every process may *broadcast* a request or message m to the others. When a request has been agreed, it is *delivered*. Atomic broadcast also solves the *consensus* problem [22, 6]. We use a variant that delivers only messages satisfying a given *external validity* condition [7].

More precisely, *Byzantine atomic broadcast with external validity* (*abv*) is defined with a validation predicate $V()$ and uses two events: *abv-broadcast*(m), to broadcast a message m to all processes, and *abv-deliver*(p, m), which delivers a message m broadcast by process p.

Predicate $V()$ validates messages. It can be computed locally by every process and ensures that a correct process only delivers messages that satisfy $V()$. More precisely, $V()$ must guarantee that when two correct processes p and q have both delivered the same sequence of messages up to some point, then p obtains $V(m) = \textsc{True}$ for any message m if and only if q also determines that $V(m) = \textsc{True}$. The standard properties of Byzantine atomic broadcast [6] (validity, no duplication, integrity, agreement, and total order) are extended by:

External validity: When a correct process delivers some message m, then $V(m) = \textsc{True}$.

In practice it may occur that not all processes agree in the above sense on the validity of a message. For instance, some correct process may conclude $V(m) = \textsc{True}$ while others find that $V(m) = \textsc{False}$. For this case it is useful to reason with the following relaxation:

Weak external validity: When a correct process delivers some message m, then at least one correct process has determined that $V(m) = \textsc{True}$ at some time between when m was broadcast and when it was delivered.

Every protocol for Byzantine atomic broadcast with external validity of which we are aware either ensures this weaker notion or can easily be changed to satisfy it.

State machine replication. Atomic broadcast is the main tool to implement state-machine replication (SMR), which executes a service on multiple processes for tolerating process faults. Throughout this work we assume that many operation requests are generated concurrently by all processes; in other words, there is request contention.

A *state machine* consists of variables and operations that transform its state and may produce some output. Traditionally, operations are *deterministic*. The state machine

functionality is defined by *execute*(), a function that takes a *state* $s \in \mathcal{S}$, initially s_0, and operation $o \in \mathcal{O}$ as input, and outputs a successor state s' and a *response* or *output value* r, such that $execute(s, o) \rightarrow (s', r)$.

A *replicated state-machine* is defined by two events: an input event *rsm-execute*(*operation*) that a process uses to invoke the execution of an operation o of the state machine; and an output event *rsm-output*(o, s, r), which is produced by the state machine. The output indicates the operation has been executed and carries the resulting state s and response r. We assume here that an operation o includes both the name of the operation to be executed and any relevant parameters.

More formally, a *replicated state machine (rsm)* receives requests that the state machine executes the operation o, in the form of *rsm-execute*(o) events; it produces *rsm-output*(o, s, r) events, to indicate that the state machine has executed an operation o, resulting in new state s, and producing response r. It is defined using standard properties [6], ensuring *agreement* on the executed sequence of operations among all correct processes; *correctness* in the sense that when a correct process has executed a sequence of operations $o_1, \ldots, o_k$, then the sequences of output states $s_1, \ldots, s_k$ and responses $r_1, \ldots, r_k$ satisfies $(s_i, r_i) = execute(s_{i-1}, o_i)$ for $i = 1, \ldots, k$; and finally, *termination*.

The standard implementation of a replicated state machine relies on an atomic broadcast protocol to disseminate the requests to all processes [33, 22].

2.3 Leader election

Implementations of atomic broadcast need to make some synchrony assumptions or employ randomization [20]. A very weak timing assumption that is also available in many practical implementations is an *eventual leader-detector oracle* [12, 22].

We define an eventual leader-detector primitive, denoted Ω, for a system with Byzantine processes. It informs the processes about one correct process that can serve as a leader, so that the protocol can progress. When faults are limited to crashes, such a leader detector can be implemented from a failure detector [12], a primitive that, in practice, exploits timeouts and low-level point-to-point messages to determine whether a remote process is alive or has crashed.

With processes acting in arbitrary ways, though, one cannot rely on the timeliness of simple responses for detecting Byzantine faults. One needs another way to determine remotely whether a process is faulty or performs correctly as a leader. Detecting misbehavior in this model depends inherently on the specific protocol being executed [17]. We use the approach of "trust, but verify," where the processes monitor the leader for correct behavior. More precisely, a leader is chosen arbitrarily, but ensuring a fair distribution among all processes (in fact, it is only needed that a correct process is chosen at least with constant probability on average, over all leader changes). Once elected, the chosen leader process gets a chance to perform well. The other processes monitor its actions. Should the leader not have achieved the desired goal after some time, they complain against it, and initiate a switch to a new leader.

This notion of "performance" depends on the specific algorithm executed by the processes, which relies on the output from the leader-detection module. Therefore, eventual leader election with Byzantine processes is not an isolated low-level abstraction, as with crash-stop processes, but requires some input from the higher-level algorithm. The Ω-*complain*(p) event allows to express this. Every process may *complain* against the current leader p by triggering this event.

Formally, a *Byzantine leader detector* (Ω) is defined with an output Ω-*trust*(p), designating process p to be trusted as leader, and an input event Ω-*complain*(p) that receives a complaint

about the performance of leader process p. Its formal properties [6] ensure that eventually, every correct process trusts some correct process; that when more than f correct processes that trust some process p complain about p, then every correct process eventually trusts a different process than p. Moreover, a correct process q does not trust a new leader unless at least one correct process has complained against the leader which q trusted before, and that eventually no two correct processes trust different processes.

It is possible to lift the output from the Byzantine leader detector to an *epoch-change* primitive, which outputs not only the identity of a leader but also an increasing *epoch number*. This abstraction divides time into a series of epochs at every participating process, where epochs are identified by numbers. The numbers of the epochs started by one particular process increase monotonically (but they do not have to form a complete sequence). Moreover, the primitive also assigns a *leader* to every epoch, such that any two correct processes in the same epoch receive the same leader. The mechanism for processes to complain about the leader is the same as for Ω.

More precisely, *Byzantine epoch-change* (Ψ) outputs events of the form Ψ-*start-epoch*(e, p), which indicate that epoch with number e and leader p starts; it also receives Ψ-*complain*(e, p) events similar to Ω. Its formal properties appears in the literature [6].

When an epoch-change abstraction is initialized, it is assumed that a default epoch with number 0 and a leader p_0 has been started at all correct processes. All "practical" BFT systems in the eventual-synchrony model starting from PBFT [10] implicitly contain an implementation of Byzantine epoch-change; this notion was described explicitly by Cachin et al. [6, Chap. 5].

3 Modular protocol

In this section we discuss the *modular* execution of replicated non-deterministic programs. Here the program is given as a black box, it cannot be changed, and the BFT system cannot access its internal data structures. Very informally speaking, if some processes arrive at a different output during execution than "most" others, then the output of the disagreeing processes is discarded. Instead they should "adopt" the output of the others, e.g., by asking them for the agreed-on state and response. When the outputs of "too many" processes disagree, the correct output may not be clear; the operation is then ignored (or, as an optimization, quarantined as non-deterministic) and the state rolled back. In this modular solution any application can be replicated without change; the application developers may not even be aware of potential non-determinism. On the other hand, the modular protocol requires that most operations are deterministic and produce almost always the same outputs at all processes; it would not work for replicating probabilistic functions.

More precisely, a *non-deterministic state machine* may output different states and responses for the same operation, which are due to probabilistic choices or other non-repeatable effects. Hence we assume that *execute* is a relation and not a deterministic function, that is, repeated invocations of the same operation with the same input may yield different outputs and responses. This means that the standard approach of state-machine replication based directly on atomic broadcast fails.

There are two ways for modular black-box replication of non-deterministic applications in a BFT system:

Order-then-execute: Applying the SMR principle directly, the operations are first ordered by atomic broadcast. Whenever a process delivers an operation according to the total order, it executes the operation. It does not output the response, however, before checking

with enough others that they all arrive at the same outputs. To this end, every process atomically broadcasts its outputs (or a hash of the outputs) and waits for receiving a given number (up to $n - f$) of outputs from distinct processes. Then the process applies a fixed decision function to the atomically delivered outputs, and it determines the successor state and the response.

This approach ensures consistency due to its conceptual simplicity but is not very efficient in typical situations, where atomic broadcast forms the bottleneck. In particular, in atomic broadcast with external validity, a process can only participate in the ordering of the next operation when it has determined the outputs of the previous one. This eliminates potential gains from pipelining and increases the overall latency.

Execute-then-order: Here the steps are inverted and the operations are executed *speculatively* before the system commits their order. As in other practical protocols, this solution uses the heuristic assumption that there is a designated *leader* which is usually correct. Thus, every process sends its operations to the leader and the leader orders them. It asks all processes to execute the operations speculatively in this order, the processes send (a hash of) their outputs to the leader, and the leader determines a unique output. Note that this value is still speculative because the leader might fail or there might be multiple leaders acting concurrently. The leader then tries to obtain a confirmation of its speculative order by atomically broadcasting the chosen output. Once every process obtains this output from atomic broadcast, it commits the speculative state and outputs the response.

In rare cases when a leader is replaced, some processes may have speculated wrongly and executed other operations than those determined through atomic broadcast. Due to non-determinism in the execution a process may also have obtained a different speculative state and response than what the leader has obtained and broadcast. This implies that the leader must either send the state (or state delta) and the response resulting from the operation though atomic broadcast, or that a process has a different way to recover the decided state from other processes.

In the following we describe Protocol *Sieve*, which adopts the approach of *execute-then-order* with speculative execution.

Protocol *Sieve*. Protocol *Sieve* runs a Byzantine atomic broadcast with weak external validity (abv) and uses a *sieve-leader* to coordinate the execution of non-deterministic operations. The leader is elected through a Byzantine epoch-change abstraction, as defined in Section 2.3, which outputs epoch/leader tuples with monotonically increasing epoch numbers. For the *Sieve* protocol these epochs are called *configurations*, and *Sieve* progresses through a series of them, each with its own sieve-leader.

The processes send all operations to the service through the leader of the current configuration, using an INVOKE message. The current leader then initiates that all processes execute the operation speculatively; subsequently the processes agree on an output from the operation and thereby *commit* the operation. As described here, *Sieve* executes one operation at a time, although it is possible to greatly increase the throughput using the standard method of *batching* multiple operations together.

The leader sends an EXECUTE message to all processes with the operation o. In turn, every process executes o *speculatively* on its current state s, obtains the speculative next state t and the speculative response r, signs those values, and sends a hash and the signature back to the leader in an APPROVE message.

The leader receives $2f + 1$ APPROVE messages from distinct processes. If the leader observes at least $f + 1$ approvals for the *same* speculative output, then it *confirms* the

operation and proceeds to committing and executing it. Otherwise, the leader concludes that the operation is *aborted* because of diverging outputs. There must be $f + 1$ equal outputs for confirming o, in order to ensure that every process will eventually learn the correct output, see below.

The leader then *abv-broadcasts* an ORDER message, containing the operation, the speculative output (t, r) for a confirmed operation or an indication that it aborted, and for validation the set of APPROVE messages that justify the decision whether to confirm or abort. During atomic broadcast, the external validity check by the processes will verify this justification.

As soon as an ORDER message with operation o is *abv-delivered* to a process in *Sieve*, o is committed. If o is confirmed, the process adopts the output decided by the leader. Note this may differ from the speculative output computed by the process. Protocol *Sieve* therefore includes the next state t and the response r in the ORDER message. In practice, however, one might not send t, but state deltas, or even only the hash value of t while relying on a different way to recover the confirmed state. Indeed, since $f + 1$ processes have approved any confirmed output, a process with a wrong speculative output is sure to reach at least one of them for obtaining the confirmed output later.

In case the leader *abv-broadcasted* an ORDER message with the decision to abort the current operation because of the diverging outputs (i.e., no $f + 1$ identical hashes in $2f + 1$ APPROVE messages), the process simply ignores the current request and speculative state. As an optimization, processes may *quarantine* the current request and flag it as non-deterministic.

As described so far, the protocol is open to a denial-of-service attack by multiple faulty processes disguising as sieve-leaders and executing different operations. Note that the epoch-change abstraction, in periods of asynchrony, will not ensure that any two correct processes agree on the leader, as some processes might skip configurations. Therefore *Sieve* also orders the configuration and leader changes using consensus (with the *abv* primitive).

To this effect, whenever a process receives a *start-epoch* event with itself as leader, the process *abv-broadcasts* a NEW-SIEVE-CONFIG message, announcing itself as the leader. The validation predicate for broadcast verifies that the leader announcement concerns a configuration that is not newer than the most recently started epoch at the validating process, and that the process itself endorses the same next leader. Every process then starts the new configuration when the NEW-SIEVE-CONFIG message is *abv-delivered*. If there was a speculatively executed operation, it is aborted and its output discarded.

The design of *Sieve* prevents uncoordinated speculative request execution, which may cause contention among requests from different self-proclaimed leaders and can prevent liveness easily. Naturally, a faulty leader may also violate liveness, but this is not different from other leader-based BFT protocols.

The details of Protocol *Sieve* are shown in Algorithms 1–2. The pseudocode assumes that all point-to-point messages among correct processes are authenticated, cannot be forged or altered, and respect FIFO order. The invoked operations are unique across all processes and *self* denotes the identifier of the executing process. Not shown in the pseudocode is a periodic concurrent check for leader progress. The process determines the age of every $o \in \mathcal{I}$ since it has been invoked and added to $\mathcal{I}$; if there are "old" operations in $\mathcal{I}$, then the process invokes Ψ-*complain*(*leader*).

The following two optimizations for *Sieve* are described in the full version [9]: First, when run in practice, every process directly executes operations and does not include the potentially large state in ORDER messages. If a *rollback* operation exists to complement *execute*, a process that has computed a diverging state can roll the operation back and obtain the state from other processes. Second, when the well-known *PBFT protocol* [10] implements

Algorithm 1 Protocol *Sieve*.

State

$\mathcal{I}$: set of invoked operations at every process — $B[p]$, for $p \in \mathcal{P}$: buffer at sieve-leader
config: sieve-config number — *leader*: sieve-leader, initially p_0
next-epoch: next sieve-config, initially $\perp$ — *next-leader*: next sieve-leader, initially $\perp$
s: current state, initially s_0 — *cur*: current operation, initially $\perp$
t: speculative state, initially $\perp$ — r: speculative response, initially $\perp$

upon invocation *rsm-execute*(o) **do**
  $\mathcal{I} \leftarrow \mathcal{I} \cup \{o\}$
  send msg. [INVOKE, *config*, o] over point-to-point link to *leader*

upon recv. msg. [INVOKE, c, o] from p **such that** $B[p] = \perp$ **and** $c =$ *config* **and** *leader* $=$ *self* **do**
  $B[p] \leftarrow o$ // buffer only the latest operation from each process

upon exists p that $B[p] \neq \perp$ **such that** *cur* $= \perp$ **and** *leader* $=$ *self* **do**
  cur $\leftarrow B[p]$
  send [EXECUTE, *config*, *cur*] over point-to-point links to all processes

upon recv. msg. [EXECUTE, c, o] from p **such that** $p =$ *leader* **and** $c =$ *config* **and** $t = \perp$ **do**
  $(t, r) \leftarrow$ *execute*(s, o)
  $\sigma \leftarrow$ *sign*$_{self}$(SPECULATE$\|$*config*$\|$*hash*$(t\|r)$)
  send msg. [APPROVE, *config*, o, *hash*$(t\|r), \sigma$] to *leader*

upon recv. $2f + 1$ msgs. [APPROVE, c_p, o_p, h_p, σ_p], each from a distinct process p, **such that**
    $c_p =$ *config* **and** $op_p =$ *cur* **and** *verify*$_p(\sigma_p$, SPECULATE$\|$*config*$\|h_p)$ **and** *leader* $=$ *self* **do**
  if there is a set $\mathcal{E}$ of $f + 1$ received APPROVE msgs. whose h_p value is equal to *hash*$(t\|r)$ **then**
    abv-broadcast([ORDER, CONFIRM, *config*, *cur*, $t, r, \mathcal{E}$])
  else
    let $\mathcal{U}$ be the set of $2f + 1$ received APPROVE msgs.
    abv-broadcast([ORDER, ABORT, *config*, *cur*, $\perp, \perp, \mathcal{U}$])

upon *abv-deliver*(p, [ORDER, *decision*, $c, o, t_c, r_c, \cdot$]) **such that** $c =$ *config* **do** // commit o
  if *leader* $=$ *self* **then**
    $B[p] \leftarrow \perp$
    cur $\leftarrow \perp$
  if $o \in \mathcal{I}$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \setminus \{o\}$
  if *decision* $=$ CONFIRM **then**
    $s \leftarrow t_c$ // adopt the agreed-on state and response, needed if $(t_c, r_c) \neq (t, r)$
    rsm-output(o, s, r_c)
  $t \leftarrow \perp$

upon Ψ-*start-epoch*(e, p) **do**
  (*next-epoch*, *next-leader*) $\leftarrow (e, p)$
  if $p =$ *self* $\wedge\ e >$ *config* **then**
    abv-broadcast([NEW-SIEVE-CONFIG, e, *self*])

upon *abv-deliver*(p, [NEW-SIEVE-CONFIG, c, p]) **do**
  (*config*, *leader*) $\leftarrow (c, p)$
  $t \leftarrow \perp$

Algorithm 2 Validation predicate $V()$ for Byzantine atomic broadcast used inside Algorithm *Sieve*.

upon invocation $V(m)$ **do**
  if $m = [\text{ORDER}, \text{DECISION}, c, o, \mathcal{M}]$ **then**
    if $\mathcal{M}$ is a set of $f+1$ msgs. of the form $[\text{APPROVE}, c_p, o_p, h_p, \sigma_p]$ **such that**
      $c_p = \mathit{config}$ **and** $o_p = o$ **and** $\mathit{verify}_p(\sigma_p, \text{SPECULATE} \| c_p \| h_p) = \text{TRUE}$ **and**
      all h_p values in $\mathcal{M}$ are equal **then**
    return TRUE
  else if $m = [\text{ORDER}, \text{ABORT}, c, o, \mathcal{M}]$ **then**
    if $\mathcal{M}$ is a set of $2f+1$ msgs. of the form $[\text{APPROVE}, c_p, o_p, h_p, \sigma_p]$ **such that**
      $c_p = \mathit{config}$ **and** $o_p = o$ **and** $\mathit{verify}_p(\sigma_p, \text{SPECULATE} \| c_p \| h_p) = \text{TRUE}$ **and**
      no $f+1$ of the h_p values in $\mathcal{M}$ are equal **then**
    return TRUE
  else if $m = [\text{NEW-SIEVE-CONFIG}, c, p]$ **then**
    if $c \leq \mathit{next\text{-}epoch}$ **and** $p = \mathit{next\text{-}leader}$ **then**
      return TRUE
  return FALSE

abv-broadcast, then the leader information and Byzantine epoch-change mechanism can be directly obtained from PBFT. This simplifies the description of *Sieve* but breaks modularity.

▶ **Theorem 1.** *Protocol* Sieve *implements a replicated state machine allowing a non-deterministic functionality* execute()*, except that demonstrably non-deterministic operations may be filtered out and not executed.*

To see why this holds, we consider first the *agreement* condition of a replicated state machine: this follows directly from the protocol and from the *abv* primitive. Every *rsm-output* event is immediately preceded by an *abv-delivered* ORDER message, which is the same for all correct processes due to *agreement* of *abv*. Since all correct processes react to it deterministically, their outputs are the same.

For the *correctness* property, note that the outputs (s_i, r_i) (state and response) resulting from an operation o must have been confirmed by the protocol and therefore the values were included in an APPROVE message from at least one correct process. This process computed the values such that they satisfy $(s_i, r_i) = \mathit{execute}(s_{i-1}, o)$ according to the protocol for handling an EXECUTE message. On the other hand, no correct process outputs anything for committed operations that were aborted, this is permitted by the exception in the theorem statement. Moreover, only operations are filtered out for which distinct correct processes computed diverging outputs, as ensured by the sieve-leader when it determines whether the operation is confirmed or aborted. In order to abort, no set of $f+1$ processes must have computed the same outputs among the $2f+1$ processes sending the APPROVE messages. Hence, at least two among every set of $f+1$ correct processes arrived at diverging outputs.

Termination is only required for deterministic operations, they must terminate despite faulty processes that approve wrong outputs. The protocol ensures this through the condition that at least $f+1$ among the $2f+1$ APPROVE messages received by the sieve-leader are equal. The faulty processes, of which there are at most f, cannot cause an abort through this. But every ORDER message is eventually *abv-delivered* and every confirmed operation is eventually executed and generates an output.

Discussion. Non-deterministic operations have not often been discussed in the context of BFT systems. The literature commonly assumes that deterministic behavior can be imposed

on an application or postulates to change the application code for isolating non-determinism. In practice, however, it is often not possible.

Liskov [27] sketches an approach to deal with non-determinism in PBFT which is similar to *Sieve* in the sense that it treats the application code modularly and uses execute-then-order. This proposal is restricted to the particular structure of PBFT, however, and does not consider the notion of external validity for *abv* broadcast.

For applications on multi-core servers, the *Eve* system [25] also executes operation groups speculatively across processes and detects diverging states during a subsequent verification stage. In case of divergence, the processes must roll back the operations. The approach taken in Eve resembles that of *Sieve*, but there are notable differences. Specifically, the primary application of Eve continues to assume deterministic operations, and non-determinism may only result from concurrency during parallel execution of requests. Furthermore, this work uses a particular agreement protocol based on PBFT and not a generic *abv* broadcast primitive.

It should be noted that *Sieve* not only works with Byzantine atomic broadcast in the model of eventual synchrony, but can equally well be run over randomized Byzantine consensus [7, 30].

4 Master-slave protocol

By adopting the *master-slave* model one can support a broader range of non-deterministic application behavior compared to the modular protocol. This design generally requires source-code access and modifications to the program implementing the functionality. In a master-slave protocol for non-deterministic execution, one process is designated as *master*. The master executes every operation first and records all non-deterministic choices. All other processes act as *slaves* and follow the same choices. To cope with a potentially Byzantine master, the slaves must be given means to verify that the choices made by the master are plausible. The master-slave solution presented here follows *primary-backup replication* [3], which is well-known to handle non-deterministic operations. For instance, if the application accesses a pseudorandom number generator, only the master obtains the random bits from the generator and the slaves adopt the bits chosen by the master. This protocol does not work for functionalities involving cryptography, however, where master-slave replication typically falls short of achieving the desired goals. Instead a cryptographically secure protocol should be used; they are the subject of Section 5.

Non-deterministic execution with evidence. As introduced in Section 3, the *execute* operation of a non-deterministic state machine is a relation. Different output values are possible and represent acceptable outcomes. We augment the output of an operation execution by adding *evidence* for justifying the resulting state and response. The slave processes may then *replay* the choices of the master and accept its output.

More formally, we now extend *execute* to *nondet-execute* as follows:

$$\textit{nondet-execute}(s, o) \;\rightarrow\; (s', r, \rho).$$

Its parameters s, o, s', and r are the same as for *execute*; additionally, the function also outputs *evidence* ρ. Evidence enables the slave processes to execute the operation by themselves and obtain the same output as the master, or perhaps only to validate the output generated by another execution. For this task there is a function

$$\textit{verify-execution}(s, o, s', r, \rho) \;\rightarrow\; \{\textsc{False}, \textsc{True}\}$$

that outputs TRUE if and only if the set of possible outputs from *nondet-execute*(s, o) contains (s', r, ρ). For completeness we require that for every s and o, when $(s', r, \rho) \leftarrow$ *nondet-execute*(s, o), it always holds *verify-execute*$(s, o, s', r, \rho) =$ TRUE.

As a basic verification method, a slave could rerun the computation of the master. Extensions to use cryptographic verifiable computation [36] are possible. Note that we consider randomized algorithms to be a special case of non-deterministic ones. The evidence for executing a randomized algorithm might simply consist of the random coin flips made during the execution.

Replication protocol. Implementing a replicated state machine with non-deterministic operations using master-slave replication does not require an extra round of messages to be exchanged, as in Protocol *Sieve*. It suffices that the master is chosen by a Byzantine epoch-change abstraction and that the master broadcasts every operation together with the corresponding evidence.

More precisely, the processes operate on top of an underlying broadcast primitive *abv* and a Byzantine epoch-change abstraction Ψ. Whenever a process receives a *start-epoch* event with itself as leader from Ψ, the process considers itself to be the master for the epoch and *abv-broadcasts* a message that announces itself as the master for the epoch. The epochs evolve analogously to the configurations in *Sieve*, with the same mechanism to approve changes of the master in the validation predicate of atomic broadcast. Similarly, non-master processes send their operations to the master of the current epoch for ordering and execution.

For every invoked operation o, the master computes $(s', r, \rho) \leftarrow$ *nondet-execute*(s, o) and *abv-broadcasts* an ORDER message containing the current epoch c and parameters o, s', r, and ρ. The validation predicate of atomic broadcast for ORDER messages verifies that the message concerns the current epoch and that *verify-execution*$(s, o, s', r, \rho) =$ TRUE using the current state s of the process. Once an ORDER message is *abv-delivered*, a process adopts the response and output state from the message as its own.

Discussion. The master-slave protocol is inspired by primary-backup replication [3], and for the concrete scenario of a BFT system, it was first described by Castro, Rodrigues, and Liskov in BASE [11]. The protocol of BASE addresses only the particular context of PBFT, however, and not a generic atomic broadcast primitive. As mentioned before, the master-slave protocol requires changes to the application for extracting the evidence that will convince the slave processes that choices made by the master are valid.

5 Cryptographically secure protocols

Security functions implemented with cryptography are more important today than ever. Replicating an application that involves a cryptographic secret, however, requires a careful consideration of the attack model. If the BFT system should tolerate that f processes become faulty in arbitrary ways, it must be assumed that their secrets leak to the adversary against whom the cryptographic scheme is employed.

Service-level secret keys must be protected and should never leak to an individual process. Two solutions have been explored to address this issue. One could delegate this responsibility to a third party, such as a centralized service or a secure hardware module at every process. However, this contradicts the main motivation behind replication: to eliminate central control points. Alternatively one may use *distributed cryptography* [16], share the keys among the processes so that no coalition of up to f among them learns anything, and perform the

cryptographic operations under distributed control. This model was pioneered by Reiter and Birman [32] and exploited, for instance, by SINTRA [4, 8] or COCA [37].

In this section we introduce a novel protocol, called *Mastercrypt*, for integrating non-deterministic cryptographic operations in a BFT system, based on the master-slave paradigm and using verifiable random functions to generate pseudorandom bits. This randomness is unpredictable and cannot be biased by a Byzantine process. In the full version [9] we furthermore review a protocol based on the well-known idea of using distributed cryptography, as discussed above. Both schemes adopt the master-slave replication protocol from the previous section.

Randomness from verifiable random functions. A *verifiable random function (VRF)* [29] resembles a pseudorandom function but additionally permits anyone to verify non-interactively that the choice of random bits occurred correctly. The function therefore guarantees correctness for its output without disclosing anything about the secret seed, in a way similar to non-interactive zero-knowledge proofs of correctness.

Efficient implementations of VRFs have not been easy to find, but the literature nowadays contains a number of reasonable constructions under broadly accepted hardness assumptions [28, 23]. In practice, when adopting the random-oracle model, VRFs can immediately be obtained from unique signatures such as ordinary RSA signatures [28].

Protocol *Mastercrypt*: Replication with cryptographic randomness from a VRF. With master-slave replication, cryptographically strong randomness secure against faulty non-leader processes can be obtained from a VRF as follows. Initially every process generates a VRF-seed and a verification key. Then it passes the verification key to a trusted entity, which distributes the n verification keys to all processes consistently, ensuring that all correct processes use the same list of verification keys. At every place where the application needs to generate (pseudo-)randomness, the VRF is used by the master to produce the random bits and all processes verify that the bits are unique. Details of this protocol can be found in the full version [9].

6 Conclusion

This paper has introduced a distinction between three models for dealing with non-deterministic operations in BFT replication: *modular* where the application is a black box; *master-slave* that needs internal access to the application; and *cryptographically secure* handling of non-deterministic randomness generation. In the past, dedicated BFT replication systems have often argued for using the master-slave model, but we have learned in the context of blockchain applications that changes of the code and understanding an application's logic can be difficult. Hence, our novel Protocol *Sieve* provides a modular solution that does not require any manual intervention. For a BFT-based blockchain platform, *Sieve* can simply be run without incurring large overhead as a defense against non-determinism, which may be hidden in smart contracts.

Acknowledgments. We thank our colleagues and the members of the IBM Blockchain development team for interesting discussions and valuable comments, in particular Elli Androulaki, Konstantinos Christidis, Angelo De Caro, Chet Murthy, Binh Nguyen, and Michael Osborne.

References

1 Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *Proc. 36th IEEE Symposium on Security & Privacy*, pages 104–121, 2015.

2 Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

3 Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

4 Christian Cachin. Distributing trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 183–192, 2001.

5 Christian Cachin, editor. *Distributed Cryptocurrencies and Consensus Ledgers (DCCL 2016)*, Online proceedings of workshop co-located with PODC, 2016. URL: `https://www.zurich.ibm.com/dccl/`.

6 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

7 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

8 Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 167–176, June 2002.

9 Christian Cachin, Simon Schubert, and Marko Vukolić. Non-determinism in Byzantine fault-tolerant replication. e-print, arXiv:1603.07351 [cs.DC], 2016. URL: `http://arxiv.org/abs/1603.07351`.

10 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

11 Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, 2003.

12 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

13 Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

14 Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. UpRight cluster services. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.

15 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 *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

16 Yvo Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, 1994.

17 Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In *Proc. 3rd European Dependable Computing Conference (EDCC-3)*, volume 1667 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 1999.

18 Sisi Duan and Haibin Zhang. Practical confidential state machine replication: How to process data privately in the cloud. In *Proc. 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

19 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

20 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, April 1985.

21 Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proc. 9th European Conference on Computer Systems (EuroSys)*, 2014.

22 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*. ACM Press & Addison-Wesley, New York, 1993.

23 Tibor Jager. Verifiable random functions from weaker assumptions. In *Proc. 12th Theory of Cryptography Conference (TCC 2015)*, volume 9015 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2015.

24 Rüdiger Kapitza, Matthias Schunter, Christian Cachin, Klaus Stengel, and Tobias Distler. Storyboard: Optimistic deterministic multithreading. In *Proc. 6th Workshop on Hot Topics in System Dependability*, 2010.

25 Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proc. 10th Symp. Operating Systems Design and Implementation (OSDI)*, 2012.

26 Ramakrishna Kotla and Michael Dahlin. High throughput Byzantine fault tolerance. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 575–584, June 2004.

27 Barbara Liskov. From viewstamped replication to Byzantine fault tolerance. In *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2010.

28 Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in Cryptology: CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 597–612. Springer, 2002.

29 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

30 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2016.

31 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

32 Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

33 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

34 Tim Swanson. Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems. Report, April 2015. URL: `http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf`.

35 Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security, Proc. IFIP WG 11.4 Workshop (iNetSec 2015)*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2016.

36 Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), February 2015.

37 Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

Flexible Paxos: Quorum Intersection Revisited

Heidi Howard[1], Dahlia Malkhi[2], and Alexander Spiegelman[3]

1 **VMware Research, Palo Alto, CA, USA; and**
University of Cambridge Computing Laboratory, Cambridge, UK
heidi.howard@cl.cam.ac.uk

2 **VMware Research, Palo Alto, CA, USA**
dahliamalkhi@gmail.com

3 **VMware Research, Palo Alto, CA, USA; and**
Viterbi Dept. of Electrical Engineering, Technion Haifa, Haifa, Israel
sashas@tx.technion.ac.il

Abstract

Distributed consensus is integral to modern distributed systems. The widely adopted Paxos algorithm uses two phases, each requiring majority agreement, to reliably reach consensus. In this paper, we demonstrate that Paxos, which lies at the foundation of many production systems, is conservative. Specifically, we observe that each of the phases of Paxos may use non-intersecting quorums. Majority quorums are not necessary as intersection is required only across phases.

Using this weakening of the requirements made in the original formulation, we propose Flexible Paxos, which generalizes over the Paxos algorithm to provide flexible quorums. We show that Flexible Paxos is safe, efficient and easy to utilize in existing distributed systems. We discuss far reaching implications of this result. For example, improved availability results from reducing the size of second phase quorums by one when the system size is even, while keeping majority quorums in the first phase. Another example is improved throughput of replication by using much smaller phase 2 quorums, while increasing the leader election (phase 1) quorums. Finally, non intersecting quorums in either first or second phases may enhance the efficiency of both.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Paxos, Distributed Consensus, Quorums

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.25

1 Introduction

Distributed consensus is the problem of reaching agreement in the face of failures. It is a common problem in modern distributed systems and its applications range from distributed locking and atomic broadcast to strongly consistent key value stores and state machine replication [36]. Lamport's Paxos algorithm [19, 20] is one such solution to this problem and since its publication it has been widely built upon in teaching, research and practice.

At its core, Paxos uses two phases, each requires agreement from a subset of participants (known as a quorum) to proceed. The safety and liveness of Paxos is based on the guarantee that any two quorums will intersect. To satisfy this requirement, quorums are typically composed of any majority from a fixed set of participants, although other quorum schemes have been proposed.

In practice, we usually wish to reach agreement over a sequence of values, known as Multi-Paxos [20]. We use the first phase of Paxos to establish one participant as a *leader* and the second phase of Paxos to propose a series of values. To commit a value, the leader must

always communicate with at least a quorum of participants and wait for them to accept the value.

In this paper, we weaken the requirement in the original protocol that all quorums intersect to require only that quorums from different phases intersect. Within each of the phases of Paxos, it is safe to use disjoint quorums and majority quorums are not necessary. We will refer to this new formulation as Flexible Paxos (FPaxos) as it allows developers the flexibility to choose quorums for the two phases, provided they meet the above requirement. FPaxos is strictly more general than Paxos and FPaxos with intersecting quorums is equivalent to Paxos.

Given that Multi-Paxos and its variants are widely deployed, such a result has wide reaching practical applications. Since the second phase of Paxos (replication) is far more common than the first phase (leader election), we can use FPaxos to reduce the size of commonly used second phase quorums. For example, in a system of 10 nodes, we can safely allow any set of only 3 nodes to participate in replication, provided that we require 8 nodes to participate when recovering from leader failure. This strategy, decreasing phase 2 quorums at the cost of increasing phase 1 quorums, is referred to in the body of the paper as *simple quorums*.

The simple quorum system reduces latency, as leaders will no longer be required to wait for a majority of participants to accept proposals. Likewise, it improves steady state throughput as disjoint sets of participants can now accept proposals, enabling better utilization of participants and decreased network load. The price we pay for this is reduced availability as the system can tolerate fewer failures whilst recovering from leader failure.

However, it is not always necessary to compromise availability for improved steady state performance. For example, in a system of 10 nodes, we can safely use any set of 5 nodes to form phase 2 quorums and any set of 6 nodes to form phase 1 quorums. This improves both the performance and availability of the second phase, without hurting availability of the first phase at all. In the paper we also illustrate that surprisingly, there are quorum systems such as grid quorums, in which FPaxos allows us to decrease the quorum sizes of both phases. Furthermore, in this quorum system, the quorums within either phase do not intersect with each other.

In the following section we outline the basic Paxos algorithm using the standard terminology. Readers who are already familiar with the algorithm should proceed directly to the next section. In §3 we describe the observation in detail and then in §4 motivate why such flexibility is useful in practice. §5 gives an informal description of why it is safe to weaken Paxos's assumption on quorum intersection. In §6 we evaluate a naïve implementation of FPaxos and demonstrate its usefulness. §7 outlines how to dynamically choose quorums and §8 relates FPaxos to the existing work in the field. A TLA+ [21] specification of the FPaxos algorithm, which has been model checked against our safety assumption, can be found in [11].

2 Paxos

We wish to decide a single value v between a set of processes. The system is asynchronous, each process may fail and the messages passed between them may be lost. Each process has one or more roles. We have three roles: the proposer, a process who wishes to have a particular value chosen, the acceptor, a process which agrees and persists decided values or the learner, a process wishing to learn the decided value.

A proposer who has a candidate value will try to propose the value to the acceptors. If a value has already been chosen, the proposer will instead learn it. The process of proposing a

value has two stages: phase 1 and phase 2, each phase requires a majority of acceptors to agree in order to proceed. We will now look at each of these stages in details:

Phase 1 – Prepare & Promise

(i) A proposer selects a unique proposal number p and sends *prepare(p)* to the acceptors.
(ii) Each acceptor receives *prepare(p)*. If p is the highest proposal number promised, then p is written to persistent storage and the acceptor replies with *promise(p',v')*. *(p',v')* is the last accepted proposal (if present) where p' is the proposal number and v' is the corresponding proposed value.
(iii) Once the proposer receives *promise* from the majority of acceptors, it proceeds to phase two. Otherwise, it may try again with higher proposal number.

Phase 2 – Propose & Accept

(i) The proposer must now select a value v. If more than one proposal was returned in phase 1 then it must choose the value associated with the highest proposal number. If no proposals were returned, then the proposer can choose its own value for v. The proposer then sends *propose(p,v)* to the acceptors.
(ii) Each acceptor receives a *propose(p,v)*. If p is equal to or greater than the highest promised proposal number, then the promised proposal number and accepted proposal is written to persistent storage and the acceptor replies with *accept(p)*.
(iii) Once the proposer receives *accept(p)* from the majority of acceptors, it learns that the value v is decided. Otherwise, it may try phase 1 again with a higher proposal number.

Paxos guarantees that once a value is decided, the decision is final and no different value can be chosen. Paxos will reach agreement provided that $\lfloor N/2 \rfloor + 1$ acceptors out of N acceptors are up and are able to communicate. Proving progress requires us to make some assumptions about the synchrony of the system, as we cannot guarantee progress in a truly asynchronous systems [7].

Usually, we wish to reach agreement over a sequence of values, which we will refer to as slots. We could use distinct instances of Paxos to decide each value in the sequence i.e. the i^{th} slot is decided by the i^{th} instance of Paxos. In practice however, we can do much better and this is referred to as Multi-Paxos.

The first phase of Paxos is independent of the value proposed for any given instance, therefore phase 1 can be executed prior to knowledge of which value to propose. Furthermore, we can aggregate phase 1 over a series of slots. We refer to a proposer who has completed phase 1 as a *leader*. To avoid loss of generality, we introduce another agent, the *client* who is the origin of values for proposal. Clients may be external to the system or co-located with other processes such as the proposers.

Figure 1 illustrates how Multi-Paxos performs in practice. The x-axis shows the number of replicas in the system, each replica performs the roles of proposer, acceptor and learner. The blue line indicates the commit latency observed by the client and the red line indicates the average request throughput. As we would expect, increasing the number of replicas will increase latency and decrease throughput. These findings are consistent with previous studies [30, 27].

A *quorum system* is the method by which we choose which sets of acceptors are able to form valid quorums. It has been observed that Paxos can be generalized to replace majority quorums with any quorum system which guarantees that any two quorums will have a non-empty intersection [20, 22]. The fundamental theorem of quorum intersection

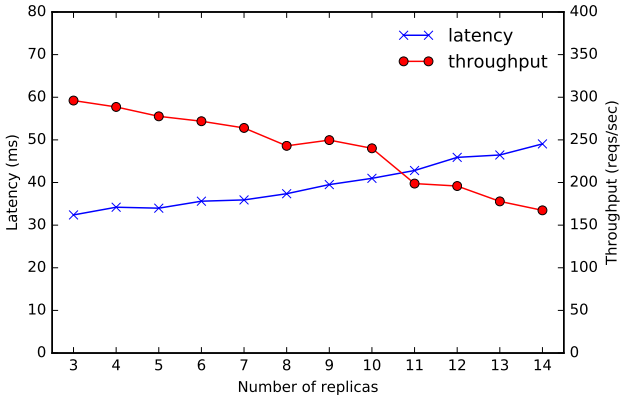


Figure 1 Performance of LibPaxos3 for varying system sizes. Details of the experimental setup are given in §6.

states that its resilience is inversely proportional to the load on (hence the throughput of) participants [32]. Therefore, with Paxos and its intersecting quorums, one can only hope to increase throughput by reducing the resilience, or vice versa. In the rest of this paper, we show that by weakening the quorum intersection requirement, we can break away from the inherent trade off between resilience and performance.

3 FPaxos

In this section, we observe that the usual description of Paxos (as given in §2) is more conservative than is necessary. To explain this observation, we will differentiate between the quorum used by the first phase of Paxos, which we will refer to as *Q1* and the quorum for second phase, referred to as *Q2*.

Paxos uses majority quorums of acceptors for both *Q1* and *Q2*. By requiring that quorums contain at least a majority of acceptors we can guarantee that there will be at least one acceptor in common between any two quorums. Paxos's proof of safety and progress is built upon this assumption that all quorums intersect.

We observe that it is only necessary for phase 1 quorums (*Q1*) and phase 2 quorums (*Q2*) to intersect. There is no need to require that *Q1*'s intersect with each other nor *Q2*'s intersect with each other. We refer to this as Flexible Paxos (FPaxos) and it generalizes the Paxos algorithm. If we allow any set of at least $\lfloor N/2 \rfloor + 1$ acceptors to form a *Q1* or *Q2* quorum in FPaxos, then FPaxos is equivalent to Paxos.

Using this observation, we can make use of many non-intersecting quorum systems. In its most straight-forward application, we can simply decrease the size of *Q2* at the cost of increasing the size of *Q1* quorums.

As we discussed earlier, the second phase of Paxos (replication) is far more frequent than the first phase (leader election) in Multi-Paxos. Therefore, reducing the size of *Q2* decreases latency in the common case by reducing the number of acceptors required to participate in replication, improves system tolerance to slow acceptors and allows us to use disjoint sets of acceptors for higher throughput. The price we pay for this is requiring more acceptors to participate when we need to establish a new leader. Whilst electing a new leader is a rare event in a stable system, if sufficient failures occur that we cannot form a *Q1* quorum, then we cannot make progress until some of the acceptors recover.

Like Paxos, the system is able to make progress provided that at least enough acceptors are up and able to communicate to form both *Q1* and *Q2* quorums. Unlike Paxos, we are able to make progress within a given phase, provided we are able to form quorums corresponding to that phase. More concretely, if sufficient failures have occurred such that a proposer can no longer form *Q1* quorums but is able to form the smaller *Q2* quorums, the system can continue to safely make progress until a new leader is required. If the acceptors recover before the current leader fails, then the system suffers no loss in availability as a result.

4 Implications

We will now consider the practical implication of observing that quorums intersection is required only between the two phases of Paxos. There already exists an extensive literature on quorum systems from the fields of databases and data replication, which can now be more efficiently applied to the field of consensus. Interesting example systems include weighted voting [9], hierarchies [16] and crumbling walls [35]. For now however, we will illustrate the utility of FPaxos by considering three naïve example quorum systems: (1) majority quorums; (2) simple quorums and (3) grid quorums.

4.1 Majority quorums

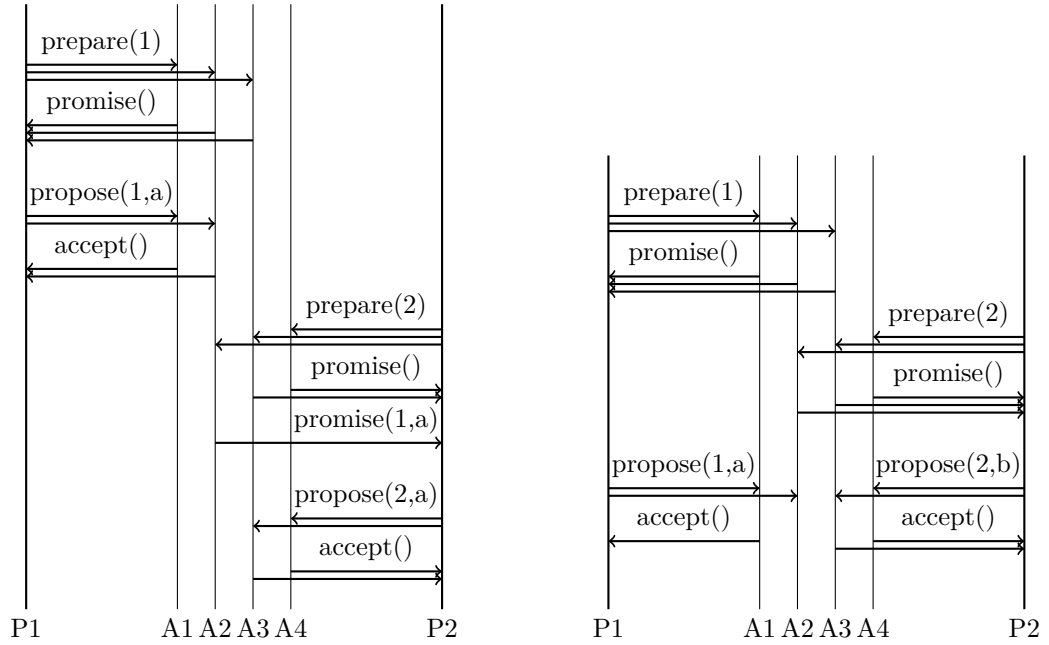
Currently, Paxos requires us to use quorums of size $N/2+1$ when the number of acceptors N is even[1]. Using our observation, we can safely reduce the size of *Q2* by one from $N/2+1$ to $N/2$ and keep *Q1* the same. Such a change would be trivial to implement and by reducing the number of acceptors required to participate in replication, we can reduce latency and improve throughput. Furthermore, we have also improved the fault tolerance of the system. As with Paxos, if at most $N/2-1$ failures occur then we are guaranteed to be able to make progress. However unlike with Paxos, if exactly $N/2$ acceptors fail and the leader is still up then we are able to continue to make progress and suffer no loss of availability.

Figure 2 shows two example traces of FPaxos with majority quorums in practice. As the system is comprised of four acceptors, FPaxos uses a majority (3 acceptors) for *Q1* but requires only two acceptors for *Q2*. In the examples, the two proposers wish to commit conflicting proposals. In Figure 2a, proposer one is first to execute FPaxos and its value a is committed. Later, proposer two executes a round of Paxos and learns the value. In Figure 2b, both proposers successfully execute the first phase of FPaxos and simultaneously submit conflicting proposed values to the disjoint sets of acceptors. Both *Q2*s will intersect with the two *Q1*s, so only one of them will be successful. In the given example, acceptor two will not accept $propose(1, a)$ as it has already promised to $prepare(2)$. The unsuccessful proposer can retry with a higher proposal number and learn the chosen value.

4.2 Simple quorums

We will use the term *simple quorums* to refer to a quorum systems where any acceptor is able to participate in a quorum and each acceptor's participation is counted equally. Simple quorums are a straightforward generalization of majority quorums. Paxos requires that all quorums intersect, and therefore, as we have previously discussed, each quorum must contain at least a strict majority of acceptors to meet this requirement.

[1] Lamport observed that majorities can be extended to include exactly half of the sets of size $N/2$ [17].



(a) FPaxos with two serial proposals.

(b) FPaxos with two concurrent proposals.

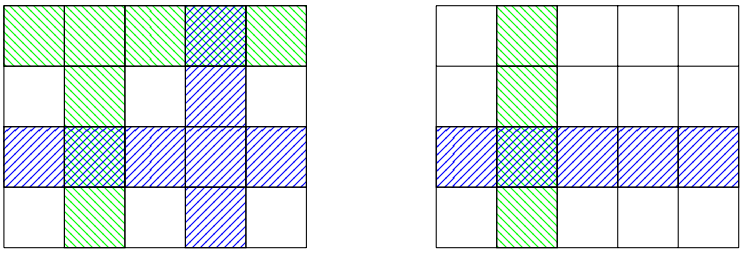
Figure 2 Sample executions of FPaxos using improved majority quorums. The system is comprised of four acceptors (A1-A4) and two proposers (P1,P2).

In contrast, FPaxos requires only that quorums from different phases intersect. Therefore, FPaxos with simple quorums must require that $|Q1| + |Q2| > N$. We know that in practice the second phase is much more common than the first phase so we allow $|Q2| < N/2$ and increase the size of $Q1$ accordingly. For a given size of *Q2* and number of acceptors N, the minimum size of our first phase quorum is $|Q1| = N - |Q2| + 1$. FPaxos will always be able to handle up to $|Q2| - 1$ failures. However, if between $|Q2|$ to $N - |Q2|$ failures occur, we can continue replication until a new leader is required.

As has been previously observed [26], we do not need to send *prepare* and *propose* messages to all acceptors, only to at least $|Q1|$ or $|Q2|$ acceptors. If any of these acceptors do not reply, then the leader can send the messages to more acceptors. This reduces the number of messages from $4 \times N$ to $(2 \times |Q1|) + (2 \times |Q2|)$. This comes at the cost of increased latency, as the leader may not choose the fastest acceptors and must retransmit when failures occur.

4.3 Grid quorums

The key limitation of simple quorums is that reducing the size of the *Q2* requires a corresponding increase in the size of *Q1* to continue to ensure intersection. Grid quorums are an example of an alternative quorum system. Grid quorums can reduce the size of *Q1* by offering a different trade off between quorum sizes, flexibility when choosing quorums and failure tolerance. Grid quorum schemes arrange the N nodes into a matrix of N_1 columns by N_2 rows, where $N_1 \times N_2 = N$ and quorums are composed of rows and columns. As with many other quorum systems, grid quorums restrict which combinations of acceptors can form valid quorums. This restriction allows us to reduce the size of quorums whilst still ensuring that they intersect.



(a) Paxos. **(b)** FPaxos.

Figure 3 Example of using a 5 by 4 grid to form quorums for a system of 20 acceptors.

Paxos requires that all quorums intersect thus one suitable grid scheme would require one row and one column to form a quorum[2]. Figure 3a shows an example *Q1* quorum and *Q2* quorum using this scheme. This would reduce the size of a quorum from the majority of N to $N_1 + N_2 - 1$. The number of failures which could be tolerated range from $MIN(N_1, N_2)$, where one node from every row or every column fails to $(N_1 - 1) \times (N_2 - 1)$, leaving only one row and one column remaining.

In FPaxos, we can safely reduce our quorums to one row of size N_1 for *Q1* and one column of size N_2 for *Q2*, examples are shown in Figure 3b. This construction is interesting as quorums from the same phase will never intersect, and may be useful in practice for evenly distributing the load of FPaxos across a group of acceptors. With simple quorums, a system cannot recover from leader failure whilst any set of $|Q2| = N/2$ acceptors have failed. Now with grid quorums, we are no longer treating all failures equally, it matters which of the acceptors have failed, not just how many have failed. Recall, that we are able to make progress in a given phase, provided we can still form a quorum for that phase. For example, let us consider if four acceptors in either of grids from Figure 3 were to fail. If these failures occur across two columns then both systems will make progress. If all the failed nodes are within one column then no progress will be made by Paxos but FPaxos will continue until a new leader is needed. Likewise, if all the nodes in a given row where to fail, FPaxos would be able to complete *Q1* and thus recover all past decisions, it can then safely fall back to a reconfiguration protocol to remove or replace the failed acceptors and continue to make progress. In practice, failures are not independent and so we can distribute acceptors across the machines, racks or even data centers to minimize the likelihood of simultaneous failure.

By way of a thought experiment, let us consider setting $N_1 = 1$ and $N_2 = N$ when using grid quorums or equivalently setting $|Q1| = N$ and $|Q2| = 1$ with simple quorums. Any single acceptor will be sufficient to form a *Q2*, however every acceptor must participate in *Q1*. In practice, this would allow all acceptors to learn the decided value in a single hop, however we would be unable to recover from leader failure until every acceptor is up.

Alternatively, let us consider setting $N_1 = N$ and $N_2 = 1$ when using grid quorums or equivalently setting $|Q1| = 1$ and $|Q2| = N$ with simple quorums. This would require every acceptor to participate in *Q2* but only a single acceptor is needed for *Q1*. If any acceptors are still up, then we can complete *Q1* locally and learn past decisions. As it has been previously observed [26, 24], such a construction allows us to tolerate f failures with only $f+1$ acceptors instead of $2f + 1$.

[2] In practice, it is sufficient to use one row plus any choice of one grid item from each row below it. The average quorum size would become $N_1 + (1/2)N_2$, although the worst case is still $N_1 + N_2 - 1$.

5 Safety

Lamport's proof of safety for Paxos does not use the full strength of the assumptions made, namely that all quorums will intersect. For the sake of completeness, in this section we outline the proof of safety for FPaxos.

For FPaxos to be safe, every decision that is reached must be final. In other words, once a value has been decided, no different value can be decided. This can be formally expressed as the following requirement:

▶ **Theorem 1.** *If value v is decided with proposal number p and v' is decided with proposal number p' then $v = v'$*

For a given value v to be decided, it must first have been be proposed. Thus the following requirement is strictly stronger:

▶ **Theorem 2.** *If value v is decided with proposal number p then for any message* propose(p',v') *where $p' > p$ then $v = v'$*

Proof is by contradiction, that is, assume $v \neq v'$. We will consider the smallest proposal number $p' > p$ for which such a message is sent.

Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be the sets of all valid phase 1 and phase 2 quorums respectively and $\mathcal{A}$ be the set of acceptors. Quorums are valid provided that:

$$\forall Q_1 \in \mathcal{Q}_1 : Q_1 \subseteq \mathcal{A} \tag{1}$$

$$\forall Q_2 \in \mathcal{Q}_2 : Q_2 \subseteq \mathcal{A} \tag{2}$$

$$\forall Q_1 \in \mathcal{Q}_1, \forall Q_2 \in \mathcal{Q}_2 : Q_1 \cap Q_2 \neq \emptyset \tag{3}$$

Equation 1 specifies that every possible phase 1 quorum is a subset of the acceptors, likewise for equation 2. Equation 3 specifies that all possible combinations consisting of a phase 1 and a phase 2 quorum will intersect in at least one acceptor.

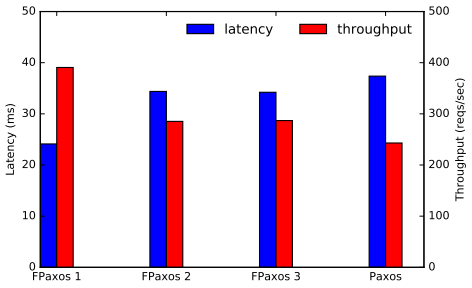
Let $Q_{p,2}$ be the phase 2 quorum used by proposal number p and $Q_{p',1}$ be the phase 1 quorum used by proposal number p'. Let $\bar{A}$ be the set of acceptors which participated both in the phase 2 quorum used by proposal number p and phase 1 quorum used by proposal number p', thus $\bar{A} = Q_{p,2} \cap Q_{p',1}$. Since $Q_{p,2} \in \mathcal{Q}_2$ and $Q_{p',1} \in \mathcal{Q}_1$ then we can use equation 3 to infer that at least one acceptor must participate in both quorums, $\bar{A} \neq \emptyset$.

Let us consider the ordering of events from the perspective of one acceptor *acc* where $acc \in \bar{A}$. It is either the case that they receive *prepare(p')* first or *propose(p,v)* first. We will consider each of these cases separately:

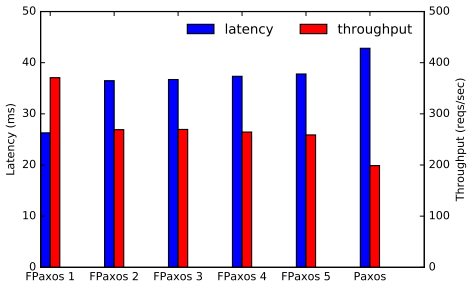
Case 1: Acceptor *acc* receives *prepare(p')* before it receives *propose(p,v)*. When *acc* receives *propose(p,v)*, its last promised proposal will be p' or higher. As $p' > p$ then it will not accept the proposal from p, however as $acc \in Q_{p,2}$ it must accept *propose(p,v)*. This is a contradiction thus it cannot be the case.

Case 2: Acceptor *acc* receives *propose(p,v)* before it receives *prepare(p')*. When *acc* receives *prepare(p')*, there are two cases. Either:

Case 2a: The last promised proposal by acceptor *acc* is already higher than p'. Then it will not accept the prepare from p', however as $acc \in Q_{p',1}$ it must accept *prepare(p')*. This is a contradiction thus it cannot be the case.



(a) Performance of FPaxos and LibPaxos3 with 5 replicas.



(b) Performance of FPaxos and LibPaxos3 with 8 replicas.

Figure 4 Throughput and average latency of FPaxos with various simple quorum sizes and LibPaxos3.

Case 2b: The last promised proposal by acceptor *acc* is less than p' then it will reply with *promise(q,v)* where $p \leq q < p'$. The value v will be the same the one *acc* accepted with p, under the minimility hypothesis on p'.

$acc \in Q_{p',1}$ therefore $promise(q, v)$ will be at least one of the responses received by the proposer of p'. If this is the only accepted value returned, then its value v will be chosen. Other proposals may also be received for members of $Q_{p',1}$. Recall that $p < p'$. For each other proposal (q', v'') received, either:

Case (i) $q' < q$: These proposal will be ignored as the proposer must choose the value associated with the highest proposal.

Case (ii) $p' < q'$: This case cannot occur as an acceptor will only reply to $prepare(p')$ when last promised is $< p'$.

Case (iii) $p < q' < p'$: For an acceptor to have accepted (q', v'') then it must have first been proposed. This is impossible by the minimality assumption on p'.

Thus the value v will be chosen, in contradiction to the assumption that $propose(p', v')$ was sent.

We model checked the formal TLA+ [21] specification of the single-valued FPaxos protocol with disjoint quorums and the requirement 2 was preserved. The FPaxos TLA+ specification, given in [11], is only a minor adaptation of the Paxos specification, given in [21].

6 Prototype

We implemented a naïve FPaxos by modifying LibPaxos3 [3], a commonly benchmarked Multi-Paxos implementation. Our modification simply generalized over the size of *Q1* and *Q2*. The simple quorums were naïvely chosen at random and messages were sent only to a quorum of nodes.

LibPaxos3 is Multi-Paxos implementation in C which uses TCP/IP for transport. For each experiment, we tested N replicas, where each replica is an acceptor, a learner and a

[3] LibPaxos3 source code `https://bitbucket.org/sciascid/libpaxos`.

possible proposer. We used request sizes of 64 bytes with 10 requests in progress at any given time and all request were first sent to the same replica. Our experiments were ran within a single linux VM with a single core and 1GB of RAM, we used mininet [4] to simulate a 10 Mbps network with 20 ms round trip time. Each test was run for 120 seconds, we discard the first and last 10 second to measure the system during its steady state.

Figure 4 show the steady state performance of Paxos and FPaxos with varying sizes of simple quorums for *Q2*. These results are as we would expect: by reducing the size of the *Q2* quorum, we send fewer messages and thus increase throughput and decrease latency.

It is worth noting that this is not the complete picture. First, FPaxos outperforms vanilla LibPaxos even with identical quorum sizes, because FPaxos sends messages only to a quorum of replicas unlike LibPaxos3 which sends messages to all replicas. When utilizing this optimization in practice, one may need to carefully trade the strategy for finding quorums in realistic settings, and consider replica failure, relative replica speeds and communication delays. Second, unlike Paxos, FPaxos with *Q2* of size 2 would not be able to elect a new leader when two acceptors have failed. On the other hand, in a system of 8 replicas, FPaxos with *Q2* of size 4 handles more failures than Paxos, decreases latency (from 42ms to 37 ms) and increases throughput (from 198 to 264 reqs/sec).

This prototype demonstrates that implementing a naïve FPaxos is trivial. We show that even a very naive implementation improves performance and we believe that systems designed for FPaxos will see far greater performance, particularly by taking advantage of using disjoint set of acceptors and smarter quorum construction techniques to improve failure tolerance. Our prototype source code and associated materials are available online[5].

7 Enhancements

We observe that the safety of FPaxos relies only on the assumption that a given *Q1* will intersect with all *Q2*s with lower proposal numbers. Therefore, we could further weaken the quorum requirements if a proposer was able to learn which *Q2*s have been used with smaller proposal numbers. We would then require only that a proposer's *Q1* intersect with these instead of all possible *Q2*s.

In order to take advantage of this, we can enhance FPaxos with a mechanism for leaders to select quorum(s) and to announce their selection. There are many ways this could be implemented, but for safety the mechanism for a leader to make its quorum selection known must be weaved carefully into the leader election protocol. Details are left out of the scope of this paper. Briefly, it would be akin to Paxos reconfiguration and achieved by adopting the principles of Vertical Paxos [24].

The implications of this enhancement can be far reaching. For example, in a system of $N = 100f$ nodes, a leader may start by announcing a fixed *Q2* of size $f + 1$ and all higher proposal numbers (and readers) will need to intersect with only this *Q2*. This allow us to tolerate $N - f$ failures. Likewise, a leader may choose a small set of $Q2$'s and announce all of them, allowing more flexibility in phase 2 at the cost of less availability in phase 1. A leader may also change its quorum selection over time using the dynamic selection mechanism.

We expect that these enhancements and others may open many new possibilities for practical system designs in the future.

[4] `http://mininet.org`
[5] `https://github.com/fpaxos`

8 Related Works

The insightful State Machine Replication (SMR) paradigm [18, 37] underlies many reliable systems, including pioneering works in distributed systems field like Viewstamped Replication [33] and Isis [3]. The Paxos algorithm provides the algorithmic solution for many production systems architected as replicated state machines. SMR must solve a core ingredient, agreement, which Dwork et al.[6] solved under minimal synchrony assumptions, and which is the basis for the single position agreement protocol (called Synod) in Paxos [19]. In the decades following its invention, the Paxos algorithm has been extensively researched: it has been explained in simpler terms [20, 41], optimized for practical systems [4, 12, 14, 34] and extended to handle reconfiguration [24] and arbitrary failures [5].

Many variants of Paxos were proposed. Cheap Paxos [26] fixes a single phase 2 quorum until a leader replacement occurs. Fast Paxos [22] has a leaderless fast-path protocol which utilizes fast-track phase 2 quorums of size $f + \lceil \frac{f+1}{2} \rceil$. Mencius [27] uses a revolving leader regime. Ring-Paxos [29, 28] applies the idea in Cheap Paxos [26] to a ring overlay using network-level multicast. Chain Replication [43] daisy-chains acceptors and collapses the two phases into one chain sweep. Generalized Paxos [23] extends state-machine replication with commutative commands. Egalitarian Paxos [31] extends Generalized Paxos with fast-track quorums whose size is $f + \lfloor \frac{f+1}{2} \rfloor$, although this can be further reduced [40]. EVE [15] optimistically concurrently agrees on commands and later resolves conflicts in case they do not commute. Corfu [2] lets the leader delegate its exclusive authority to any proposer in order to yield better parallelism. There are many other variants; a comprehensive taxonomy of Paxos variants is given in [42]. These previous works were built on the foundations presented in the pioneering protocols [33, 3, 6, 19], and focused on enhacing them in order to achieve better performance. Our new observation revisited the foundations and generalized them; it is completely orthogonal and can be integrated into previous protocols as well as to real production systems in order to further improve performance.

The SMR reconfiguration problem was addressed in several previous works. Some use consensus commands to agree on next configurations [33, 25, 24], whereas others use the first phase to determine which quorum (out of a fix set of quorums) will be used in the second phase [26, 29]. A general framework for reconfiguration that separates the steady state agreement mechanism from the reconfiguration event appears in [24]. Reconfiguration for other fault tolerant services was also previously investigated, e.g., in [10, 1, 13, 39, 8]. As discussed in §7, the ideas in these works can be adopted in order to enhance FPaxos into a reconfigurable and dynamic system.

To the best of our knowledge, we are the first to prove and implement this generalization of Paxos. During the preparation of this publication, Sougoumarane independently made the same observation on which this work is based and released a blog post summarizing [38] it for the systems community.

9 Conclusion

In this paper we have described FPaxos, a generalization of the widely adopted Paxos algorithm, which no longer requires that quorums from the same Paxos phase intersect. We believe this result has wide ranging consequences.

Firstly, over the last two decades Multi-Paxos has been widely studied, deployed and extended. Generalizing existing systems to use FPaxos should be quite straightforward. Exposing replication (phase 2) quorum size to developers would allow them to choose their own trade off between failure tolerance and steady state latency.

Secondly, by no longer requiring replication quorums to intersect, we have removed an important limit on scalability. Through smart quorum construction and pragmatic system design, we believe a new breed of scalable, resilient and performant consensus algorithms is now possible.

Acknowledgements. We wish to thank the following people for their feedback: Jean Bacon, Jon Crowcroft, Stephen Dolan, Matthew Grosvenor, Anil Madhavapeddy, Sugu Sougoumarane and Igor Zabloctchi.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011. doi:10.1145/1944345.1944348.

2 Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, December 2013.

3 Ken Birman and Thomas Joseph. *Exploiting virtual synchrony in distributed systems*, volume 21. ACM, 1987.

4 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1298455.1298487.

5 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI'99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=296806.296824.

6 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

7 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

8 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 140–153. Springer, 2015.

9 David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP'79, pages 150–162, New York, NY, USA, 1979. ACM. doi:10.1145/800215.806583.

10 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

11 Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexable paxos: tlaplus specification. URL: https://github.com/fpaxos/fpaxos-tlaplus.

12 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

13 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

14 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.

15 Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: execute-verify replication for multi-core servers. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, 2012.

16 Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. Comput.*, 40(9):996–1004, September 1991. doi:10.1109/12.83661.

17 Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978. doi:10.1016/0376-5075(78)90045-4.

18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

19 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

20 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 2001.

21 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

22 Leslie Lamport. Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research, 2005.

23 Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

24 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*, PODC'09, pages 312–313, New York, NY, USA, 2009. ACM. doi:10.1145/1582716.1582783.

25 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

26 Leslie Lamport and Mike Massa. Cheap paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN'04, pages 307–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=1009382.1009745.

27 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855741.1855767.

28 Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

29 Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 527–536. IEEE, 2010.

30 P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 527–536, June 2010. doi:10.1109/DSN.2010.5544272.

31 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Op-*

erating Systems Principles, SOSP'13, pages 358–372, New York, NY, USA, 2013. ACM. doi:10.1145/2517349.2517350.

32 Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, April 1998. doi:10.1137/S0097539795281232.

33 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

34 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

35 David Peleg and Avishai Wool. Crumbling walls: A class of practical and efficient quorum systems. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 120–129, New York, NY, USA, 1995. ACM. doi:10.1145/224964.224978.

36 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.

37 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

38 Sugu Sougoumarane. A more flexible paxos. [Online; accessed 13-Aug-2016]. URL: http://ssougou.blogspot.com/2016/08/a-more-flexible-paxos.html.

39 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. *OPODIS 2015*, 2015.

40 P. Sutra and M. Shapiro. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 255–264, Oct 2011. doi:10.1109/SRDS.2011.38.

41 Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015. doi:10.1145/2673577.

42 Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015. doi:10.1145/2673577.

43 Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

Relaxed Byzantine Vector Consensus*†

Zhuolun Xiang[1] and Nitin H. Vaidya[2]

1 **Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, USA**
xiangzl@illinois.edu

2 **Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Illinois, USA**
nhv@illinois.edu

Abstract

Byzantine vector consensus requires that non-faulty processes reach agreement on a decision (or output) that is in the convex hull of the inputs at the non-faulty processes. Recent work has shown that, for n processes with up to f Byzantine failures, when the inputs are d-dimensional vectors of reals, $n \geq \max(3f+1, (d+1)f+1)$ is the tight bound for synchronous systems, and $n \geq (d+2)f+1$ is tight for approximate consensus in asynchronous systems.

Due to the dependence of the lower bound on vector dimension d, the number of processes necessary becomes large when the vector dimension is large. With the hope of reducing the lower bound on n, we propose relaxed versions of Byzantine vector consensus: k-relaxed Byzantine vector consensus and (δ, p)-relaxed Byzantine vector consensus. k-relaxed consensus only requires consensus for projections of inputs on every subset of k dimensions. (δ, p)-relaxed consensus requires that the output be within distance δ of the convex hull of the non-faulty inputs, where distance is defined using the L_p-norm. An input-dependent δ allows the distance from the non-faulty convex hull to be dependent on the maximum distance between the non-faulty inputs.

We show that for k-relaxed consensus with $k > 1$, and for (δ, p)-relaxed consensus with constant $\delta \geq 0$, the bound on n is identical to the bound stated above for the original vector consensus problem. On the other hand, when $k = 1$ or δ depends on the inputs, we show that the bound on n is smaller when $d \geq 3$. Input-dependent δ may be of interest in practice. In essence, input-dependent δ scales with the spread of the inputs.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Byzantine consensus, vector inputs, relaxed validity conditions

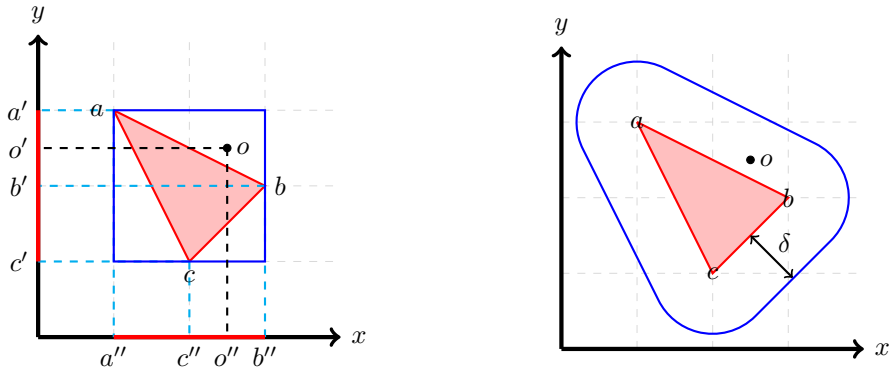
Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.26

1 Introduction

Byzantine vector consensus requires that non-faulty processes reach agreement on a decision (or output) that is in the convex hull of the inputs at the non-faulty processes. This paper considers Byzantine consensus in a complete network consisting of n processes of which up to f processes may be Byzantine faulty [5]. Recent work has shown that when the inputs are d-dimensional vectors of reals, $n \geq \max(3f+1, (d+1)f+1)$ is the tight bound on the number of processes n to be able to achieve exact Byzantine consensus in a synchronous system [6, 11, 7].

* This research is supported in part by National Science Foundation award 1421918. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agency or the U.S. government.

† A brief announcement summarizing the results appeared at SPAA 2016.



(a) k-relaxed convex hull.

(b) (δ, p)-relaxed convex hull.

Figure 1 Illustrations for relaxed convex hull.

Exact Byzantine vector consensus is defined as follows:

▶ **Definition 1** (Exact Byzantine vector consensus (exact BVC))**.** Exact BVC must satisfy the following three conditions [11, 7]:

1. *Agreement:* The decision (or output) vector at all the non-faulty processes must be identical.
2. *Validity:* The decision vector at each non-faulty process must be in the convex hull of the input vectors at the non-faulty processes.
3. *Termination:* Each non-faulty process must terminate after a finite amount of time.

Due to the dependence of the above bound on vector dimension d, the number of processes necessary becomes large when the vector dimension is large. With the hope of reducing the lower bound on n, we consider *relaxed* versions of Byzantine vector consensus: k-relaxed Byzantine vector consensus and (δ, p)-relaxed Byzantine vector consensus (we often refer to these as k-relaxed consensus and (δ, p)-relaxed consensus, respectively). For (δ, p)-relaxed consensus, we consider two formulations: constant δ and input-dependent δ, respectively.

For brevity, this paper first focuses on exact vector consensus in synchronous systems, and its relaxations. Analogous results for relaxations of approximate Byzantine vector consensus in asynchronous systems are summarized in Section 6.

Our relaxed versions of the Byzantine vector consensus problem are defined by replacing the convex hull in the above Validity condition by a *relaxed convex hull*, in particular, *k-relaxed convex hull* or *(δ, p)-relaxed convex hull* as defined later in Section 4.1 and 5.1. Intuitively, a k-relaxed convex hull, illustrated in Figure 1a, consists of points which are contained in the convex hull of the projections of inputs at non-faulty processes, where the projections are taken on every subset of k dimensions. (δ, p)-relaxed convex hull, illustrated in Figure 1b, consists of points within the distance δ of the convex hull of the inputs of non-faulty processes, where distance is defined using the L_p-norm. Formal definitions of the two *relaxations* of a convex hull are presented in Section 4.1 and 5.1.

The original vector consensus problem (Definition 1) is obtained as a special case of the two relaxed versions, by choosing $k = d$ in k-relaxed consensus, and $\delta = 0$ in (δ, p)-relaxed consensus. Also note that, when $d = 1$, the inputs are scalar, and all the L_p norms are identical. For the case of $d = 1$, (δ, p)-relaxed consensus with constant $\delta > 0$ is similar to a consensus problem addressed in prior work [2]. In particular, [2] showed (for scalar inputs) that even if a valid output is allowed to be outside the range of non-faulty inputs by up to δ,

the number of processes necessary to achieve consensus cannot be smaller than $3f+1$. Thus, our work on (δ, p)-relaxed consensus extends the above prior work [2] to higher dimensions.

With the exception of Section 6, the rest of the paper assumes that the system is synchronous. For synchronous systems, we obtain the following results:

- We show that $n \geq \max(3f+1, (d+1)f+1)$ is the tight bound on n for k-relaxed consensus for $1 < k \leq d$. That is, when $k > 1$, the relaxation does not reduce the number of processes necessary. However, when $k = 1$, $n \geq 3f+1$ is the tight bound for all dimensions d. Thus, $k = 1$ significantly reduces the tight bound on n when d is large.
- For any *constant* $\delta \geq 0$ that is *independent* of the inputs, we show that $n \geq \max(3f+1, (d+1)f+1)$ remains the tight bound on n for (δ, p)-relaxed consensus. That is, the relaxation does not lower the bound.
- For values of δ specified as a function of the inputs of the non-faulty processes, we show that (δ, p)-consensus can be achieved using a smaller number of processes than the above bound for the case of constant δ. We establish a relationship between n and an achievable value of δ. For instance, for $f = 1$ and $d \geq 3$, we show that $(\frac{e_{max}}{n-2}, 2)$-consensus and $(\frac{e_{min}}{2}, 2)$-consensus is achievable with $4 \leq n \leq d+1$ processes, where e_{max} (e_{min}) is the maximum (minimum) distance between the inputs of any two fault-free processes. We also obtain results for some other values of f, n and p, and propose a conjecture for the open cases.

  Section 6 summarize analogous results for asynchronous systems.

2 Related Work

Lamport, Shostak and Pease [5] developed the initial results on Byzantine fault-tolerant agreement. As noted above, for the special case of $d = 1$, our (δ, p)-relaxed consensus is similar to the so-called "$(\epsilon, \delta, \gamma)$-agreement" problem addressed in prior work [2]. *Byzantine vector consensus* (BVC) (also called *multidimensional* consensus) was introduced by Mendes and Herlihy [6] and Vaidya and Garg [11]. Tight bounds on number of processes n for Byzantine vector consensus have been obtained for synchronous [11] and asynchronous [6, 11] systems both, when the network is a complete graph. A necessary condition and a sufficient condition for *iterative* BVC in incomplete graphs were derived by Vaidya [10], however, there is a gap between these necessary and sufficient conditions.

A related problem of Convex Hull Consensus was introduced by Tseng and Vaidya [9], wherein the goal for the non-faulty processes is to try to learn the largest possible subset of the convex hull of the non-faulty inputs. For this problem, fault-tolerant algorithms have been proposed for asynchronous systems under crash faults [9] and Byzantine faults [8], respectively.

Herlihy et al. [3] introduce the (d, ϵ)-*solo approximate agreement* problem in the context of a d-solo execution model, which yields the message-passing model and the traditional shared memory model as special cases. For (d, ϵ)-solo approximate agreement, the inputs are d-dimensional vectors of reals, and the outputs must be in the convex hull of the inputs. Up to d processes may potentially choose as their outputs any arbitrary points in the convex hull of all inputs (not necessarily approximately equal to each other), while each remaining process must choose as its output a point within distance ϵ of the convex hull of the outputs of these d processes (all outputs must be within the convex hull of the inputs). Although Herlihy et al. [3] only consider crash failures, their problem formulation can be easily extended to the Byzantine fault model. The relaxed consensus formulations considered in our work are distinct from (d, ϵ)-solo agreement.

3 Notations and Terminology

The total number of processes is n, with up to f processes suffering Byzantine failures. The processes are numbered as $1, 2, \cdots, n$. Each process can communicate directly with all the processes (i.e., the network is a complete graph). The input at each process is a d-dimensional vector of reals, $d \geq 1$. We view each input as a *column* vector. Dimensions (or coordinates) of a d-dimensional vector are indexed as $1, 2, \cdots, d$. Transpose of vector u is denoted u^T. We often view a *vector* as a *point* in an appropriate space. The i-th element (or i-th coordinate) of vector v is denoted as $v[i]$. The set $\{1, 2, \cdots, d\}$ is denoted as $[1, d]$. For $u, v \in \mathbb{R}^d$, distance $\|u - v\|_p$ using L_p-norm is defined as $\|u - v\|_p = \left(\sum_{i=1}^{d} |u[i] - v[i]|^p\right)^{1/p}$. By definition, L_∞-norm is defined as $\|u - v\|_\infty = \max_{i=1,\cdots,d}(|u[i] - v[i]|)$.

A multiset may potentially contain repetitions of an element. Let $\mathcal{H}(S)$ denote the convex hull of a multiset S. For a multiset Y, when we write $X \subseteq Y$, X is a multiset in which frequency of each element is no greater than its frequency in multiset Y. The size of the multiset S, denoted $|S|$, is the number of elements in S, counting all repetitions. For a multiset Y with $|Y| \geq f$, define $\Gamma(Y)$ as

$$\Gamma(Y) = \bigcap_{X \subseteq Y, |X| = |Y| - f} \mathcal{H}(X) \tag{1}$$

In Section 4, we consider (δ, p)-relaxed Byzantine vector consensus, and Section 5 focuses on k-relaxed Byzantine vector consensus.

4 (δ, p)-Relaxed Byzantine Vector Consensus

4.1 Definition

To be able to define (δ, p)-relaxed consensus, we first define a relaxed notion of a convex hull.

▶ **Definition 2.** For $\delta \geq 0$ and $p \geq 1$, (δ, p)-relaxed convex hull $H_{(\delta,p)}$ of $S \subseteq \mathbb{R}^d$ is

$$H_{(\delta,p)}(S) = \{u \mid \|u - v\|_p \leq \delta,\ v \in \mathcal{H}(S)\}\,.$$

As an example, see Figure 1b. In the figure a, b, c are 2-dimensional inputs of three non-faulty processes. Let $p = 2$. The red triangle in the figure is the convex hull of a, b, c, while the area within the blue curve is the (δ, p)-relaxed convex hull of a, b, c, where δ is the length shown in the figure.

(δ, p)-relaxed consensus must satisfy the *Agreement* and *Termination* conditions stated in Section 1, and the *relaxed* validity condition below.

> **(δ, p)-relaxed validity:** The decision vector at each non-faulty process must be in the (δ, p)-*relaxed convex hull* of the set of input vectors at the non-faulty processes.

We consider two ways to specify δ: (i) δ may be specified as a constant (Section 4.3), or (ii) δ may be input-dependent, in particular, specified as a function of the distance between the inputs at the non-faulty processes (Section 4.4).

Consider Figure 1b again, with a, b, c being the inputs of non-faulty processes. Instead of the δ shown in the figure, suppose that we choose an input-dependent δ. Specifically, let $\delta =$ minimum distance between non-faulty inputs. Then in this example, δ will equal the

length of segment bc, resulting in a larger *relaxed* convex hull than that encompassed by the blue curve in Figure 1b. On the other hand, if we were to have $a = b = c$, then the minimum distance would be 0, resulting in δ being 0 as well. In this manner, we can use input-dependence to scale δ with the "spread" of the non-faulty inputs.

4.2 Preparation

As noted before, for brevity, the following discussion assumes that the systems is synchronous. Results for asynchronous systems are summarized in Section 6.

The following lemma can be proved easily.

▶ **Lemma 3.** *Solving (δ, p)-Relaxed BVC implies solving (δ', p)-Relaxed BVC where $\delta \leq \delta'$. That is, a necessary condition for (δ', p)-Relaxed BVC is also necessary for (δ, p)-Relaxed BVC, and a sufficient condition for (δ, p)-Relaxed BVC is also sufficient for (δ', p)-Relaxed BVC.*

The proof of Lemma 3 is provided in our full version [12].

We make some simple observations about two special cases of (δ, p)-relaxed BVC.

- When $\delta = 0$, the problem formulation become identical to the original exact BVC problem (Definition 1). Thus, $n \geq \max(3f+1, (d+1)f+1)$ is the necessary and sufficient condition in this special case.
- When $\delta = \infty$, the validity condition for (∞, p)-relaxed consensus is vacuous, allowing the processes to choose any fixed vector in $\mathbb{R}^d$ as the output (e.g., the processes may always choose the all-0 vector as their output and still satisfy the validity condition with $\delta = \infty$).

4.3 Results for constant δ

▶ **Theorem 4.** *$n \geq \max(3f + 1, (d + 1)f + 1)$ is necessary and sufficient for (δ, p)-Relaxed Exact BVC in a synchronous system, where $0 < \delta < \infty$ and $1 \leq p$.*

Proof. When $d = 1$, the inputs are scalar, and all the L_p norms are identical. For the case of $d = 1$, (δ, p)-relaxed consensus is similar to a problem that was addressed in prior work [2]. For this case, it can be shown similarly that $n \geq 3f + 1$ is necessary and sufficient. Therefore, in the rest of the proof, we assume $d \geq 2$.

Sufficiency: Due to the equivalence of the original Exact BVC and $(0, p)$-Relaxed Exact BVC, for $d \geq 2$ and $1 \leq p$, $n \geq (d + 1)f + 1$ is sufficient for $(0, p)$-Relaxed Exact BVC. Then by Lemma 3, this condition is also sufficient for (δ, p)-Relaxed BVC where $0 < \delta < \infty$.

Necessity: We first prove that $n \geq d + 2$ is necessary for $f = 1$ and $p = \infty$. The proof is by contradiction. Suppose that $n = d + 1$ and (δ, ∞)-Relaxed Exact BVC is achievable using a certain algorithm.

Let us suppose that exactly one process is Byzantine faulty, but the faulty process correctly follows any specified algorithm. Due to this restricted behavior, it is possible for all the processes to correctly learn the input of all the other processes. If we can show that $d + 1$ processes are insufficient despite the above constraint on the faulty process, then $d + 1$ are insufficient when arbitrary behaviors are allowed for the faulty process. Hereafter, we assume that all the processes follow the specified algorithm.

Let the i^{th} column of the following $d \times (d+1)$ matrix S be an input vector of the i^{th} process, where $x > 2d\delta$.

$$S = \begin{pmatrix} x & 0 & \cdots & \cdots & 0 & 0 \\ 0 & x & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & x & 0 & 0 \\ 0 & \cdots & \cdots & 0 & x & 0 \end{pmatrix}$$

For $1 \leq i \leq d$, the i-th coordinate of the i-th input is x, and the rest of the coordinates are 0. The $d+1$-th input is all-0. Let Y denote the set of all inputs specified in matrix S. If N is the set of non-faulty processes, then the output must be in $H_{(\delta,\infty)}(N)$. However, since the identity of any faulty process is unknown, the decision vector must be in

$$\bigcap_{T \subseteq Y, |T|=|Y|-f} H_{(\delta,\infty)}(T)$$

where $f = 1$.

Now we consider different choices of T:

- *Observation 1:* Consider T as the set of all inputs except the input of process i, $1 \leq i \leq d$. Then the i^{th} element of each of the d inputs in T is 0. Therefore the i^{th} element of all the vectors in $H_{(\delta,\infty)}(T)$ – and consequently in the output – must be less than or equal to δ due to the definition of (δ, ∞)-relaxed validity.
- *Observation 2:* Consider T as the set of all inputs except the input of process $(d+1)$. The vectors in $H_{(\delta,\infty)}(T)$ are within distance δ (where the distance is measured using the L_∞ norm) of the convex hull $\mathcal{H}(T)$. In each convex combination of elements in T used to obtain the convex hull $\mathcal{H}(T)$, at least one of the weights must be $\geq \frac{1}{d}$. Hence at least one element of each vector in $\mathcal{H}(T)$ must be $\geq \frac{x}{d}$. Thus, at least one element of each vector in $H_{(\delta,\infty)}(T)$ – and consequently the output – must be $\geq \frac{x}{d} - \delta > \delta$ (recall that $x > 2d\delta$).

Thus, Observation 1 and 2 contradict each other, proving that $n = d+1$ is not sufficient for $f = 1$.

For $f > 1$, we can use the well-known simulation approach to show $n = (d+1)f$ is not sufficient [5]. Therefore, $n \geq (d+1)f + 1$ is necessary for (δ, ∞)-*Relaxed Exact BVC* with $f > 1$, completing the proof.

Now, for any vector x, $\|x\|_\infty \leq \|x\|_p$, for $1 \leq p < \infty$ [4]. Therefore, we have

$$H_{(\delta,p)} \subseteq H_{(\delta,\infty)}\,.$$

Then, the argument above for (δ, ∞)-consensus would imply that $n \geq (d+1)f + 1$ is also necessary for (δ, p)-*Relaxed Exact BVC*. ◀

Theorem 4 shows a disappointing result. Specifically, when δ is a constant, the relaxed validity condition of (δ, p)-relaxed consensus does not yield a reduction in the number of processes necessary to solve the problem.

On the other hand, as shown in Section 4.4 below, the tight bound on n can be lower when δ is input-dependent. In general, the results for input-dependent δ are more challenging to prove than the results presented above.

4.4 Results for input-dependent δ

In contrast to **constant** δ, if the relaxation parameter δ depends on the non-faulty inputs themselves, then the (δ, p)-Relaxed Exact BVC problem may be solvable when $3f+1 \leq n \leq (d+1)f$. Interpreting a d-dimensional vector as a point in the d-dimensional Euclidean space, define E_+ as the set of edges between the inputs at the non-faulty processes in any given execution. The input-dependent δ will be defined below using the edge set E_+. In particular, we prove the following results for $p = 2$. An extension to general L_p-norm is provided in our full version [12].

▶ **Theorem 5.** *Let E_+ denote the set of edges between the inputs at non-faulty processes. When* (i) *$f = 1$ and $4 \leq n \leq d+1$, or* (ii) *$f \geq 2$ and $3f+1 \leq n = (d+1)f$, $(\widehat{\delta}, 2)$-relaxed consensus is achievable where*

$$\widehat{\delta} = \frac{\max_{e \in E_+} \|e\|_2}{\lfloor \frac{n}{f} \rfloor - 2}.$$

Observe that $\widehat{\delta}$ depends on the inputs of non-faulty processes – however, for brevity, our notation $\widehat{\delta}$ does not make that dependence explicit. If inputs of all non-faulty processes happen to be identical, then $\widehat{\delta}$ would be 0. On the other hand, if the non-faulty inputs are far apart, then $\widehat{\delta}$ would be accordingly larger (larger $\widehat{\delta}$ allows the output to be farther away from the convex hull of the non-faulty inputs).

The cases considered in Theorem 5 satisfy the constraint $3f+1 \leq n \leq (d+1)f$. It is well-known that at least $3f+1$ processes are necessary for scalar Byzantine consensus. A similar argument, as presented in our full version [12], shows that $n \geq 3f+1$ is also necessary for (δ, p)-relaxed consensus with input-dependent δ for all d. Secondly, if $n > (d+1)f$, then $\delta = 0$ is achievable (i.e., the "unrelaxed" problem is solvable). Hence the constraint $3f+1 \leq n \leq (d+1)f$ is meaningful. Note that this constraint can only be met when $d \geq 3$.

The above theorem considers two special cases. When $f = 1$, the above expression becomes $\widehat{\delta} = \frac{\max_{e \in E_+} \|e\|_2}{n-2}$, and when $n = (d+1)f$, it becomes $\widehat{\delta} = \frac{\max_{e \in E_+} \|e\|_2}{d-1}$. The above theorem does not make any claims about the case when $f \geq 2$ and $3f+1 \leq n < (d+1)f$. We conjecture that $\widehat{\delta}$ specified in the theorem is achievable even in these cases. For the case of $f = 1$, we are able to strengthen the above result, as stated next.

▶ **Theorem 6.** *When $f = 1$ and $4 \leq n \leq d+1$, $(\widehat{\delta}, 2)$-relaxed consensus is achievable where*

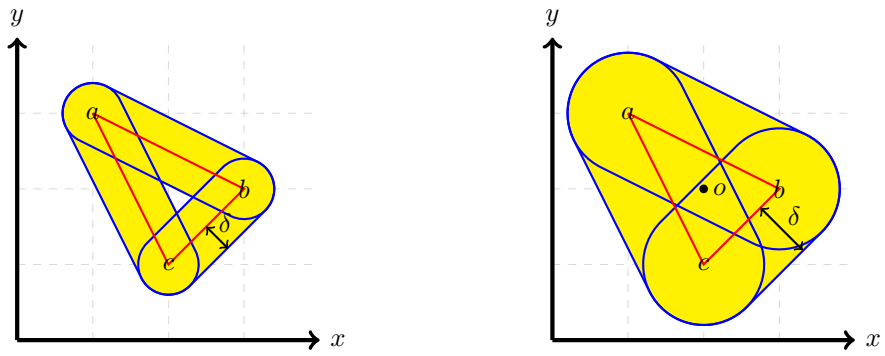
$$\widehat{\delta} = \min\left(\frac{\min_{e \in E_+} \|e\|_2}{2}, \frac{\max_{e \in E_+} \|e\|_2}{n-2}\right)$$

Our proof of Theorems 5 and 6 is constructive. We show that the *Relaxed BVC* (R-BVC) algorithm presented below satisfies the claims in these theorems. While algorithm R-BVC is quite straightforward, our key contribution here is to show that the algorithm achieves $(\widehat{\delta}, 2)$-relaxed consensus, as claimed in Theorems 5 and 6.

Algorithm R-BVC

Let v_i denote the d-dimensional input at process i, $1 \leq i \leq n$.

- *Step 1:* Each process i performs a Byzantine broadcast of its input v_i. Byzantine broadcast of each element of the vector v_i can be performed separately by using any Byzantine broadcast algorithm, such as [5]. $n \geq 3f+1$ suffices for the correctness of Byzantine



(a) Smaller δ. **(b)** Larger δ.

Figure 2 Impact of δ on $\Gamma_{(\delta,2)}(\{a,b,c\})$.

broadcast in a completely connected network. At the completion of Step 1, each process will receive a multiset $S = \{a_i \mid 1 \leq i \leq n\}$, where for a non-faulty process i, $a_i = v_i$, the input of process i, and for a faulty process j, a_j may be any arbitrary point in the d-dimensional space. Importantly, all non-faulty processes obtain identical multiset S. The points in S received from non-faulty processes are said to *non-faulty inputs*, and the remaining points are said to be *faulty inputs*.

- *Step 2:* Each process determines the smallest value δ such that

$$\Gamma_{(\delta,2)}(S) = \bigcap_{T \subseteq S, |T| = |S| - f} H_{(\delta,2)}(T)$$

is non-empty, and for this value of δ, the process deterministically chooses a point in $\Gamma_{(\delta,2)}(S)$ as its output. All processes use the same deterministic function to choose the output from $\Gamma_{(\delta,2)}(S)$.

Let $\delta^*(S)$ denote the smallest value of δ for which $\Gamma_{(\delta,2)}(S)$ is non-empty. When the set S is clear from context, we will abbreviate $\delta^*(S)$ simply as δ^*. δ^* is well-defined, because by choosing δ sufficiently large, it is always possible to ensure that $\Gamma_{(\delta,2)}(S)$ is non-empty. This is illustrated in Figures 2a and 2b for the case when $d = 2$, $S = \{a, b, c\}$, and $f = 1$. Note that the cylinder around each red edge is the $(\delta, 2)$-relaxed convex hull of the endpoints of that edge. With the smaller value of δ used in Figure 2a, $\Gamma_{(\delta,2)}(S)$ is empty, but it is non-empty in Figure 2b with a larger value of δ.

Recall that, for $(\widehat{\delta}, 2)$-relaxed consensus, the validity condition requires the output to be in the relaxed convex hull $H_{(\widehat{\delta},2)}$ of the non-faulty inputs. However, since a non-faulty process does not know which processes are faulty, and let $\delta^*(S)$ denote the smallest value of δ for which $\Gamma_{(\delta,2)}(S)$ is non-empty. $\widehat{\delta}$ depends on inputs of non-faulty processes, it is not possible for the non-faulty processes to compute $\widehat{\delta}$ explicitly. Instead, the above algorithm chooses an output in $\Gamma_{(\delta^*,2)}(S)$ (where $\delta^*(S)$ is defined above). We will show that $\delta^*(S) \leq \widehat{\delta}$. By Lemma 3, this implies that $(\widehat{\delta}, 2)$-relaxed consensus is achieved.

4.5 Proof of Theorem 6

We now prove Theorem 6 stated previously.

Proof. Note that the discussion below makes frequent references to set S of inputs collected in Step 1 of algorithm R-BVC. Recall that E_+ is the set of edges between non-faulty inputs in S. Let E denote the set of edges between *all* points in S.

Consider set $S = \{a_1, \cdots, a_n\}$ obtained in Step 1 of algorithm R-BVC. If the n points in S are **not** affinely independent, then the $n-1$ vectors in the set $\{a_i - a_n \mid i \neq n, 1 \leq i \leq n\}$ are **not** linearly independent. In this case, it is easy to show that $\delta^*(S) = 0$. The proof of this claim is presented in our full version [12]. Thus, in this case, $(0, 2)$-relaxed consensus (i.e., "unrelaxed" version) is achievable.

Hereafter, we assume that the n points in S **are** affinely independent, thus, the $n-1$ vectors in $\{a_i - a_n \mid i \neq n, 1 \leq i \leq n\}$ are linearly independent.

Recall that Theorem 6 assumes $f = 1$. We consider two cases separately: (I) $n = d+1$ and (II) $4 \leq n < d+1$.

4.5.1 Case I: $n = d + 1$

We begin with a useful lemma.

▶ **Lemma 7.** *Let $d \geq 2$. When the points in $S = \{a_1, \cdots, a_{d+1}\}$ are affinely independent, let r be the radius of the inscribed sphere of the simplex(contained within the simplex and tangent to each of the simplex's faces) formed by the points in S. Let E denote the set of edges between every pair of vertices of this simplex, and let π_k denote the facet of the simplex that contains $\{a_i \mid i \neq k,\ 1 \leq i \leq d+1\}$ (i.e., all vertices except a_k), $k = 1, \cdots, d+1$. Then π_k itself is a simplex in a $(d-1)$-dimensional subspace. Let r_k be the radius of the $(d-1)$-dimensional inscribed sphere of π_k in this $(d-1)$-dimensional subspace. Then,*

$$r = \delta^*(S), \tag{2}$$

$$r < \min_{1 \leq k \leq d+1} r_k, \quad \text{and} \tag{3}$$

$$r < \frac{\max_{e \in E} \|e\|_2}{d} \tag{4}$$

The proof of the three claims in the above lemma are presented in our full version [12].

Proof that $\delta^*(S) < \frac{\min_{e \in E_+} \|e\|_2}{2}$

This claim will be proved here by induction for any simplex in dimensions ≥ 2. Consider a simplex in 2 dimensions. Then the simplex is simply a triangle, and it can be easily shown that the radius of its inscribed sphere is $<$ half the length of the shortest edge in the triangle. Our full version [12] presents the proof of this claim.

Now, suppose that, for every simplex of dimension $k \geq 2$, the radius of its inscribed sphere is $<$ half the length of its shortest edge, and consider a simplex A of dimension $k+1$. Equation (3) in Lemma 7 then implies that the radius of the inscribed sphere of A is also $<$ half the length of the shortest edge in A. Inductively, for the simplex formed by S, this proves that $r < \frac{\min_{e \in E} \|e\|_2}{2}$. Since $E_+ \subseteq E$, it then follows that $r < \frac{\min_{e \in E_+} \|e\|_2}{2}$. Then by equation (2) of Lemma 7, we have that

$$\delta^*(S) < \frac{\min_{e \in E_+} \|e\|_2}{2}. \tag{5}$$

Proof that $\delta^*(S) < \frac{\max_{e \in E_+} \|e\|_2}{d-1}$

Without loss of generality, assume that process 1 is faulty, and thus $a_1 \in S$ is the only faulty input in S. Let π_1 be the facet of the simplex formed by the points in $S - \{a_1\}$, and r_1 be the radius of $(d-1)$-dimensional inscribed sphere of π_1. Observe that π_1 is

isomorphic to a simplex in $d-1$ dimensions. By equation (4) of Lemma 7 (when applied to $d-1$ dimensional points), we have $r_1 < \frac{\max_{e\in E'} \|e\|_2}{d-1}$, where E' is the set of edges between the input corresponding to π_1 (i.e., inputs in $S - \{p_1\}$). Now, since p_1 is the input of the only faulty process, it follows that E' equals E_+ (i.e., the set of edges between non-faulty inputs). Thus, $r_1 < \frac{\max_{e\in E_+} \|e\|_2}{d-1}$. By equations (2) and (3) of Lemma 7, we then have $\delta^*(S) = r < r_1 < \frac{\max_{e\in E_+} \|e\|_2}{d-1}$.

This result, in conjunction with (5) proves that

$$\delta^*(S) \quad < \quad \min\left(\frac{\min_{e\in E_+} \|e\|_2}{2}, \frac{\max_{e\in E_+} \|e\|_2}{d-1}\right). \tag{6}$$

Note that, since $n = d+1$, $d-1$ equals $n-2$, thus (6) proved Theorem 6 when $n = d+1$ and $f = 1$.

4.5.2 Case II: $4 \leq n < d+1$

Since the vectors in $\{a_i - a_n \mid 1 \leq i < n\}$ are linearly independent, these vectors form a $n-1$ dimensional subspace W (where $n-1 < d$). Then we can find a projection matrix P that projects these d-dimensional vectors into a $(n-1)$-dimensional space, while preserving the distances between the points in $S = \{a_1, \cdots, a_n\}$. Then the n points $Pa_1, \cdots, Pa_n$ form a simplex in a $(n-1)$-dimensional subspace. By the results in Case I, and substituting d by $n-1$, the claim follows in Case II.

Thus, we have proved that algorithm R-BVC achieves $(\delta^*(S), 2)$-relaxed consensus, where $\delta^*(S) < \widehat{\delta} = \min\left(\frac{\min_{e\in E_+} \|e\|_2}{2}, \frac{\max_{e\in E_+} \|e\|_2}{n-2}\right)$. Then, by Lemma 3, R-BVC also achieves $(\widehat{\delta}, 2)$-relaxed consensus, proving Theorem 6. ◀

4.6 Proof of Theorem 5

We presented Theorem 5 earlier. The proof of Theorem 5 is significantly more complex than Theorem 6. For lack of space, we only sketch the proof here. A complete proof is presented in our full version [12].

Proof Sketch. Theorem 5 considers the case when $3f+1 \leq n = (d+1)f$. Since the case of $f = 1$ is provided by Theorem 6, here we focus on case (ii) where $n = (d+1)f$ and $f \geq 2$.

For brevity below, we often refer to $\delta^*(S)$ simply as δ^*. Let P_i be the subsets of S of size $(n-f) = df$, $i = 1, \cdots, \binom{n}{f}$. Recall that δ^* denotes the smallest value of δ for which $\Gamma_{(\delta,2)}(S)$ is non-empty. Equivalently, there exists a point in $\Gamma_{(\delta,2)}(S)$, whose largest distance to any $\mathcal{H}(P_i)$ is minimized compared to any other points, and this largest distance is exactly δ^*. Hence, it is easy to see that $\delta^* = \min_{p\in\mathbb{R}^d}\left(\max_{i=1,\cdots,\binom{n}{f}} dist(p, \mathcal{H}(P_i))\right)$. Let $p_0 \in \arg\min_{p\in\mathbb{R}^d}\left(\max_{i=1,\cdots,\binom{n}{f}} dist(p, \mathcal{H}(P_i))\right)$. Also, suppose that m of the P_i's are distance exactly δ^* from p_0; let Q_j, $j = 1\cdots, m$, denote these m subsets (P_i's). The rest of the subsets are at distance $> \delta^*$. Now we consider the following two cases.

- *Case 1:* $1 \leq m \leq d$:
  In this case, we can show that $\delta^* = 0$ by contradiction. Suppose $\delta^* > 0$. We can then move p_0 towards a suitable direction to find a new point p' which is closer to all $\mathcal{H}(P_i)$ for all i, when compared to p_0, contradicting the definition of p_0. Thus, $\delta^* = 0$. Therefore, $(0, 2)$-relaxed consensus is achievable, and the theorem is true in Case 1.

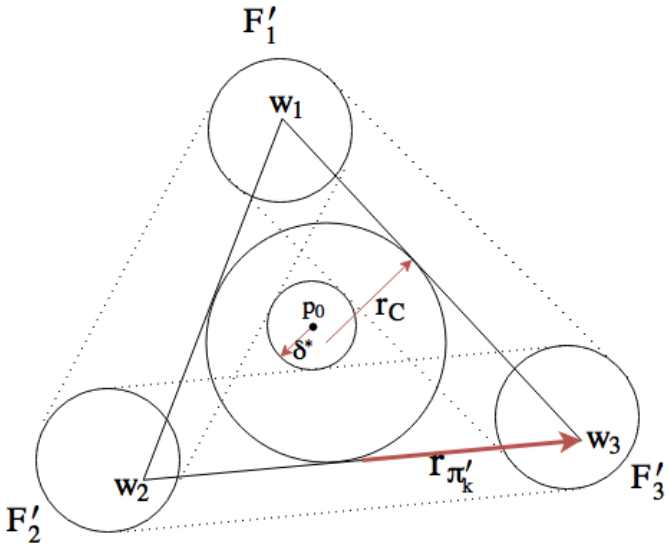


Figure 3 Illustration of the proof.

Case 2: $m \geq d+1$:

In this case, we can first show that there exist $d+1$ distinct sets $Q'_1, \cdots, Q'_{d+1}$ in $\{Q_j\}$, such that $\bigcap_{i=1}^{d+1} \mathcal{H}(Q'_i) = \emptyset$. Let $F'_i = S - Q'_i$, $i = 1, \cdots, d+1$. We can show that these F'_i's are disjoint, and form a partition of S.

Then, we can show the following three claims to prove that p_0 is contained in the simplex A formed by $W = \{w_i \mid 1 \leq i \leq d+1\}$, with each $w_i \in F'_i$, as illustrated in the Figure 3 (here $d = 2$ as an simple example).

▶ **Claim 8.** *Consider a set Z of size $d+1$ consisting of one point each in F'_i. Then the $d+1$ points in Z are affinely independent, and $A = \mathcal{H}(Z)$ is a simplex in d-dimensions.*

▶ **Claim 9.** $\mathcal{H}(S) - \bigcup_{i=1}^{d+1} \mathcal{H}(Q'_i) \subseteq A$.

▶ **Claim 10.** *p_0 is contained in the simplex A formed by $W = \{w_i \mid 1 \leq i \leq d+1\}$, i.e., $p_0 \in \mathcal{H}(W)$.*

In Figure 3, $A = \mathcal{H}(Z)$ is the triangle (which is a simplex in dimension 2) formed by w_1, w_2, w_3 (Claim 8). $\mathcal{H}(S) - \bigcup_{i=1}^{d+1} \mathcal{H}(Q'_i)$ is the dotted triangle in the center, which is contained in the triangle A (Claim 9). It is clear from the figure that p_0 is contained in A (Claim 10).

Then, by Claim 10, p_0 is contained in the simplex A formed by $W = \{w_i \mid 1 \leq i \leq d+1\}$ with each $w_i \in F'_i$, where $|F'_i| = f$ and $\cup_{i=1}^{d+1} F'_i = S$.

Let π'_k denote the facet of simplex A that contains $\{w_i \mid i \neq k,\ 1 \leq i \leq d+1\}$. Let r_A be the radius of the sphere inscribed in the simplex A. Since p_0 is contained in the simplex A, it can be shown that $r_A \geq \delta^*$. Then considering the distribution of faulty inputs among sets F'_i. There are two cases:

1. There exists k, $1 \leq k \leq d+1$, such that all the faulty inputs are contained in F'_k. Then π'_k is the convex hull of a subset of non-faulty inputs. By equation (3) of Lemma 7, we have $r_A < r_{\pi'_k}$. Then by equation (4) of Lemma 7, $r_{\pi'_k} < \frac{\max_{e \in E'} \|e\|_2}{d-1}$, where E' is the set of edges between vertices of π'_k. Since π'_k consists of only non-faulty inputs,

we have $r_{\pi'_k} < \frac{\max_{e\in E'}\|e\|_2}{d-1} \leq \frac{\max_{e\in E_+}\|e\|_2}{d-1}$. Hence $\delta^* \leq r_A < r_{\pi'_k} < \frac{\max_{e\in E_+}\|e\|_2}{d-1}$. The relationship between δ^* and $r_{\pi'_k}$ is shown intuitively in Figure 3.

2. There does not exist k such that all the faulty inputs are contained in F'_k. Then we can take one non-faulty input from each F'_i and form a simplex C which contains p_0. By equation (4) in Lemma 7, we know that $r_C < \frac{\max_{e\in E''}\|e\|_2}{d}$, where E'' is the set of edges between the vertices of C. Since vertices of C are all non-faulty inputs, we have $r_C < \frac{\max_{e\in E''}\|e\|_2}{d} \leq \frac{\max_{e\in E_+}\|e\|_2}{d}$. Hence, $\delta^* \leq r_C < \frac{\max_{e\in E_+}\|e\|_2}{d} < \frac{\max_{e\in E_+}\|e\|_2}{d-1}$. The relationship between δ^* and r_C is also shown intuitively in Figure 3.

Then, by Lemma 3, R-BVC achieves $(\widehat{\delta}, 2)$-relaxed consensus, proving Theorem 5 ◀

4.7 A Conjecture

While Theorem 6 covers all the interesting cases for $f = 1$, Theorem 5 leaves some cases undecided for $f \geq 2$. We conjecture that the claim of Theorem 5 is also true for $f = 2$ and $3f + 1 \leq n < (d + 1)f$.

5 k-Relaxed Byzantine Vector Consensus

5.1 Definition

To be able to define k-relaxed consensus, we first introduce other definitions.

▶ **Definition 11** (D-projection)**.** Let $D = \{d_1, d_2, \cdots, d_k\}$ where $1 \leq d_i < d_j \leq d$ for $1 \leq i < j \leq k$. For $u \in \mathbb{R}^d$ define projection $g_D(u) = v$ where $v \in \mathbb{R}^k$ and $v[i] = u[d_i]$. For multiset S consisting of points in $\mathbb{R}^d$, define $g_D(S) = \{g_D(u) \mid u \in S\}$.

Thus, D-projection g_D defined above projects a given vector on the specified set of k coordinates. When a set is provided as an argument, g_D returns D-projection of each vector in that set. While g_D is not a one-to-one function, with an abuse of terminology, we will define its inverse.

▶ **Definition 12** (Inverse of D-projection)**.** For $v \in \mathbb{R}^k$, define $g_D^{-1}(v) = U$ where $U \subset \mathbb{R}^d$, such that $u \in U$ if and only if $g_D(u) = v$. For multiset S consisting of points in $\mathbb{R}^k$, define $g_D^{-1}(S) = \bigcup_{v\in S} g_D^{-1}(v)$.

For example, suppose that $d = 4$, $D = \{1, 3\}$, $u = (7, -4, -2, 0)^T$, and $v = (7, -2)^T$. Then $g_D(u) = (7, -2)^T$, and $g_D^{-1}(v) = \{(7, a, -2, b)^T \mid a, b \in \mathbb{R}\}$.

▶ **Definition 13.** The k-relaxed convex hull H_k of $S \subset \mathbb{R}^d$ is defined as

$$H_k(S) = \{u \mid g_D(u) \in \mathcal{H}(g_D(S)), \forall D \in \mathcal{D}_k\}$$

where $\mathcal{D}_k = \{D \mid D \subseteq [1, d],\ |D| = k\}$. Equivalently,

$$H_k(S) = \bigcap_{D\in\mathcal{D}_k} g_D^{-1}(\mathcal{H}(g_D(S)))\,.$$

As an example, see Figure 1a. In this example, we consider inputs a, b, c of dimension 2. Let $k = 1$. The red triangle is the convex hull of a, b, c, while the area within the blue rectangle is the k-relaxed convex hull of a, b, c (for $k = 1$). Any point in H_k, o for instance, is contained in the convex hull formed by projections of a, b, c on each dimension (because $k = 1$). o' and o'' are projections of o on each of the dimensions. Clearly, o' is contained in

the convex hull formed by a', b', c', and o'' is contained in the convex hull formed by a'', b'', c'' (projections of a, b, c).

Now we define k-relaxed consensus. In particular, k-relaxed consensus must satisfy the *Agreement* and *Termination* conditions in Section 1, and the *relaxed* validity condition below.

> ***k*-relaxed validity:** The decision vector at each non-faulty process must be in the *k-relaxed convex hull* of the set of input vectors at the non-faulty processes.

5.2 Results

This section presents our key results on k-Relaxed Byzantine Vector Consensus. As noted before, results for asynchronous systems are summarized in Section 6.

We begin with some simple observations about a few special cases of relaxed BVC.

- For *k-relaxed BVC* with $k = d$, the problem formulation becomes identical to the original exact BVC problem(Definition 1). Thus, $n \geq \max(3f + 1, (d + 1)f + 1)$ is the necessary and sufficient condition in these special cases.
- When $k = 1$, k-relaxed consensus (i.e., 1-relaxed consensus) can be achieved using any Byzantine consensus algorithm for scalar inputs (such as [1]). In particular, the processes perform d instances of a scalar Byzantine consensus algorithm. The input of any process j for the i-th instance is the i-th coordinate of process j's d-dimensional input for 1-relaxed consensus. Each process j uses the output of the i-th instance of scalar Byzantine consensus above to be the i-th coordinate of its output vector for 1-relaxed consensus. It is easy to verify that this solution correctly achieves 1-relaxed consensus. Thus, the tight bound on n for $k = 1$ is identical to the well-known bound for scalar Byzantine consensus, namely, $n \geq 3f + 1$ [5].

▶ **Theorem 14.** *$n \geq (d + 1)f + 1$ is necessary and sufficient for k-relaxed consensus in a synchronous system when $2 \leq k \leq d - 1$.*

The proof of Theorem 14 is provided in our full version [12]. Analogous to Theorem 4, the above theorem also shows a negative result.

6 Results for Asynchronous Systems

In this section, we briefly present our results for relaxed Byzantine vector consensus in asynchronous systems.

Approximate Byzantine vector consensus in asynchronous systems must satisfy the *Validity* and *Termination* conditions stated in Definition 1, and the *ϵ-Agreement condition* below.

> ***ϵ*-Agreement:** The decision (or output) vectors at any two non-faulty processes must be within distance ϵ of each other, where $\epsilon > 0$.

From previous studies, $n \geq (d+2)f+1$ is necessary and sufficient for the above approximate Byzantine consensus [6, 11, 7]. As we will see soon, the results for relaxed Byzantine vector consensus in asynchronous systems are analogous to those of synchronous systems.

6.1 (δ, p)-Relaxed Byzantine Vector Consensus

(δ, p)-relaxed approximate Byzantine vector consensus in asynchronous systems must satisfy the *ϵ-Agreement condition* above, the *(δ, p)-relaxed Validity* condition in Section 4.1 and the *Termination* condition.

With Constant δ

▶ **Theorem 15.** *$n \geq (d+2)f + 1$ is necessary and sufficient for (δ, p)-Relaxed Approximate BVC in an asynchronous system, where $0 < \delta < \infty$ and $1 \leq p$.*

The proof of Theorem 15 is analogous to the one of Theorem 4, and is provided in our full version [12].

The condition on number of processes $n \geq (d+2)f + 1$ remains unchanged when we relax the validity condition, compared with the original Byzantine vector consensus problem.

With Input-dependent δ

For brevity, we only present the results. The algorithm is provided in our full version [12].

▶ **Theorem 16.** *Suppose $(\widehat{\delta}, p)$-Relaxed Exact BVC is achievable where*

$$\widehat{\delta} = \kappa(n, f, d, p) \max_{e \in E_+} \|e\|_p$$

where $\kappa(n, f, d, p)$ is a finite constant that may depend on number of processes n, number of failures f, dimension of the inputs d and L_p norm. E_+ is the set of edges between pairs of non-faulty inputs in S.

Then $(\widehat{\delta}, p)$-Relaxed Approximate BVC is achievable where

$$\widehat{\delta} = \kappa(n - f, f, d, p) \max_{e \in E_+} \|e\|_p$$

where $\kappa(n, f, d, p)$, S and E_+ are defined above.

The proof of Theorem 16 is provided in our full version [12].

By Theorem 16 and the previous results in Section 4.4, (δ, p)-relaxed approximate consensus can be solved with fewer number of processes. Moreover, we established a formula between the size of feasible δ for solving synchronous case and that for solving asynchronous case. Namely, we can infer the feasible δ for asynchronous case from that for synchronous case.

6.2 k-Relaxed Byzantine Vector Consensus

k-relaxed approximate Byzantine vector consensus in asynchronous systems must satisfy the *ϵ-Agreement condition* above, the *k-relaxed Validity* condition in Section 5.1 and the *Termination* condition.

For $k = 1$, $n \geq 3f + 1$ is necessary and sufficient, and for $k = d$, $n \geq (d+2)f + 1$ is necessary and sufficient. Bounds for $d = 1, 2$ are included in the above results.

▶ **Theorem 17.** *$n \geq (d+2)f + 1$ is necessary and sufficient for $2 \leq k \leq d-1$ in k-Relaxed Approximate BVC in an asynchronous system.*

The proof of Theorem 17 is analogous to the one of Theorem 14, and is provided in our full version [12].

Similar to Theorem 15 and Theorem 14, the result for k-consensus in asynchronous systems is also negative.

7 Summary

This paper studies *k-relaxed Byzantine vector consensus*, and (δ, p)-*relaxed Byzantine vector consensus* with constant and input-dependent δ both. For k-relaxed consensus and (δ, p)-relaxed consensus with constant δ, the tight necessary and sufficient condition on number of processes is shown to be identical to that for the original ("unrelaxed") consensus problem. For the case of input-dependent δ, we show that the problem is solvable with fewer processes.

References

1. Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, 1992.
2. Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. *Easy impossibility proofs for distributed consensus problems*. Springer, 1990.
3. Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Computing in the presence of concurrent solo executions. In *LATIN 2014: Theoretical Informatics*, pages 214–225. Springer, 2014.
4. Gottfried Köthe. *Topological vector spaces*. Springer, 1983.
5. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.
6. Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, pages 391–400. ACM, 2013.
7. Hammurabi Mendes, Maurice Herlihy, Nitin H. Vaidya, and Vijay K. Garg. Multidimensional agreement in byzantine systems. *Distributed Computing*, 28(6):423–441, 2015.
8. Lewis Tseng and Nitin H. Vaidya. Byzantine convex consensus: An optimal algorithm. *arXiv preprint arXiv:1307.1332*, 2013.
9. Lewis Tseng and Nitin H. Vaidya. Asynchronous convex hull consensus in the presence of crash faults. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 396–405. ACM, 2014.
10. Nitin H. Vaidya. Iterative byzantine vector consensus in incomplete graphs. In *Distributed Computing and Networking*, pages 14–28. Springer, 2014.
11. Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, pages 65–73. ACM, 2013.
12. Zhuolun Xiang and Nitin H Vaidya. Relaxed byzantine vector consensus. *arXiv preprint arXiv:1601.08067*, 2016.

m-Consensus Objects Are Pretty Powerful*

Ammar Qadri

University of Toronto, Toronto, Canada
ammar.qadri@mail.utoronto.ca

Abstract

A recent paper by Afek, Ellen, and Gafni introduced a family of deterministic objects $O_{m,k}$, for $m, k \geq 2$, with consensus numbers m such that, for each $k \geq 2$, $O_{m,k}$ is computationally less powerful than $O_{m,k+1}$ in systems with at least $mk+m+k$ processes. This paper gives a wait-free implementation of $O_{m,k}$ from $(m+1)$-consensus objects and registers in systems with any finite number of processes. In order to do so, it introduces a new family of objects which helps us to understand the power of m-consensus among more than m processes.

1998 ACM Subject Classification E.1 Distributed Data Structures, F.1.2 Parallelism and concurrency

Keywords and phrases Deterministic Consensus Hierarchy, Wait-free Implementation, Tournament

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.27

1 Introduction

The consensus problem is a fundamental problem in distributed computing that has long been used to categorize the computational powers of shared objects. Herlihy [5] defined the *consensus number* of an object, which is the largest number of processes for which wait-free consensus can be achieved using only instances of the object and registers. Thus an object O with consensus number n cannot be implemented in a wait-free manner by an object O' with consensus number $n' < n$ in a system with more than n' processes. The *consensus hierarchy* classifies objects by their consensus numbers.

Herlihy also proved that any object can be implemented by n-consensus objects and registers (and, hence, by any objects with consensus number n) in systems with n or fewer processes. However, the relative computational powers of objects with the same consensus number n in systems of more than n processes is not entirely understood.

Afek, Gafni, Tromp, and Vitányi [3] showed that test-and-set objects, which have consensus number 2, can be implemented from 2-consensus objects and registers in a system with any finite number of processes. Afek, Weisberger, and Weisman [4] and Afek, Gafni, and Morrison [2] proved that this is also true for other well-known objects of consensus number 2, such as fetch-and-increment objects, swap objects, and stacks. It was conjectured [4] that this is true for any object with consensus number 2. This is known as the *Common2 Conjecture*. More generally, the *Consensus Hierarchy Conjecture* asserts that for $n \geq 2$, every shared object of consensus number $n' \leq n$ has a wait-free implementation from n-consensus objects and registers in every system with a finite number of processes.

Rachman [6] disproved the Consensus Hierarchy Conjecture for nondeterministic objects. He showed that for any m' and m, with $m' \geq m \geq 1$, there exists a nondeterministic object

* This work has been generously supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

X with consensus number m, such that X cannot be implemented using only m'-consensus objects and registers in systems with at least $2m' + 1$ processes. This means that the consensus hierarchy, at least on its own, is not very useful in characterizing the computational powers of nondeterministic objects.

Afek, Ellen, and Gafni [1] proved that the Consensus Hierarchy Conjecture does not even hold for the class of deterministic objects. They introduced the $O_{m,k}$ object, for $m, k \geq 2$, and showed that each $O_{m,k}$ object has consensus number m, but cannot be implemented (in a non-blocking manner) from m-consensus objects and registers in any system with at least $km + k - 1$ processes. More surprisingly, they showed that an $O_{m,k+1}$ object cannot be implemented (in a non-blocking manner) from $O_{m,k}$ objects and registers in any system with at least $mk + m + k$ processes, meaning that $O_{m,2}, O_{m,3}, \ldots$ is an infinite sequence of objects with increasing computational power, all with consensus number m.

This paper determines the relationship between $O_{m,k}$ objects and $(m + 1)$-consensus objects. Section 4 presents a wait-free implementation of every $O_{m,k}$ object from $(m + 1)$-consensus objects and registers among any finite number of processes. Thus, $O_{m,k}$ objects lie strictly between m-consensus objects and $(m+1)$-consensus objects in terms of computational power. This provides additional understanding of the consensus hierarchy for deterministic objects and is a step towards a characterization of their computational power.

For our implementation, we introduce a new family of deterministic objects, Q_r, for $r \geq 0$, which serves as a crucial synchronization mechanism. The Q_r object is a generalization of the test-and-set object: it has an operation which allows the first process that performs it to win and all subsequent processes to fail. In addition, there is another operation that returns the identity of the winner to the first r processes who perform it after the object has been won. Section 3 formally defines the Q_r object, proves that it has consensus number $r + 2$, and proves that it has a wait-free implementation from $(r + 2)$-consensus objects and registers in any system with a finite number of processes. It is hoped that the Q_r object will be a useful tool for determining the power of $(r + 2)$-consensus objects in systems of more than $r + 2$ processes.

2 Model

We consider an asynchronous shared memory system, with n processes, $p_1, p_2, \ldots, p_n$, which communicate by applying operations to shared objects. A *step* by any process consists of an operation it is performing on a shared object as well as the response received from the operation.

A *configuration* of the shared memory system consists of the current state of every process, as well as the current value of every shared object. An *initial configuration* is a configuration where all processes and shared objects are in one of their initial states. An *execution* is specified by an alternating sequence of configurations and steps by processes, beginning at an initial configuration, such that if a configuration C is immediately followed by a step s of process p_i, which is immediately followed by a configuration C', then p_i performing step s in configuration C yields the configuration C'.

An *implementation* of a sequentially specified shared object O from a set of shared base objects is given by providing an algorithm for each operation of O in which processes only access the base objects. The implementation is *linearizable* if, in any execution $\mathcal{E}$, we can order each completed operation as well a subset of incomplete operations on O such that the results of each operation in this order are consistent with the sequential specification of O, and, moreover, an operation that completes before another operation begins comes

earlier in the order. Equivalently, the implementation is linearizable if, in any execution $\mathcal{E}$, we can assign a (distinct) linearization point to each completed operation as well as a subset of incomplete operations such that the linearization point of every operation is within the execution interval of that operation and the results of each operation in the order induced by the linearization points are consistent with the sequential specification of O. The implementation is *wait-free* if every process that performs any operation of O completes the operation in a finite number of its own steps. The implementation is *non-blocking* if, from every configuration of any execution, there is some process that completes its operation in a finite number of steps.

Consensus can be solved by n processes, $p_1, p_2, \ldots, p_n$, if there exists an algorithm such that, if every process p_i is assigned some input x_i, each process outputs at most one value, satisfying the following conditions:

- *validity*: If p_i outputs y_i, then there exists some j such that $y_i = x_j$.
- *agreement*: If p_i outputs y_i and p_j outputs y_j, then $y_i = y_j$.
- *termination*: Every process that takes sufficiently many steps outputs a value.

The m-consensus object is a shared object with a single operation PROPOSE that takes one argument. The first m PROPOSE operations to an m-consensus object return the argument of the first PROPOSE operation. We say that this value is the *decision* of the m-consensus object. All other operations return $\perp$.

The test-and-set object is a shared object with a single operation T&S, which returns *true* for the first operation and *false* for all subsequent operations.

The fetch-and-increment object is a shared object and has a single operation F&I, which returns the value $a + i - 1$ to the i^{th} operation, where a was the initial value of the object.

Throughout this paper, we assume that the initial value of every fetch-and-increment object is 1 and the initial value of every register is $\perp$.

3 The Q_r object

We now formally define the Q_r object and explain how it can be used together with registers to solve consensus among $r + 2$ processes. Then, in Section 3.1, we prove that it can be implemented from $(r + 2)$-consensus objects and registers in a system with any finite number of processes.

The Q_r object, for $r \geq 0$, has two operations, COMPETE and QUERY, with the following specifications:

- The first COMPETE operation returns *true* and the process that performs it is called the *winner* of the object. All subsequent COMPETE operations return $false$.
- The first r QUERY operations after the first COMPETE operation return the *id* of the winner. All other QUERY operations return $\perp$.

If processes perform only COMPETE operations, then a Q_r object behaves like a test-and-set object. In particular, Q_0 is equivalent to a test-and-set object and, thus, has consensus number 2. The additional power of a Q_r object for $r > 0$ is due to the r QUERY operations that can be used to determine the winner.

The Q_r object can solve consensus among $r + 1$ processes: Each process announces its input in a single writer register before calling COMPETE on a shared Q_r object. The winner decides its own input, whereas the r losers call QUERY to find the identity of the winner, and then read and decide the winner's announced value.

It is only slightly more difficult to see that the Q_r object can solve consensus among $r + 2$ processes. As before, each of the processes, $p_1, p_2, \ldots, p_{r+2}$, first announces its input in

Algorithm 1 A solution to $(r+2)$-consensus.

Shared variables:
q is a Q_r object
$announce[1 \dots r+2]$ is an array of registers

```
p1: function PROPOSE(v) by process p_i for i ∈ {1, ..., r+2}
p2:     announce[i].WRITE(v)
p3:     if q.COMPETE() then
p4:         return v
p5:     end if
p6:     announce[i].WRITE(⊥)
p7:     winner ← q.QUERY()
p8:     if winner ≠ ⊥ then
p9:         return announce[winner].READ()
p10:    end if
p11:    for j ← 1 to r+2 do
p12:        value ← announce[j].READ()
p13:        if value ≠ ⊥ then
p14:            return value
p15:        end if
p16:    end for
p17: end function
```

a single writer register. Next, processes perform COMPETE on a Q_r object and the winner decides its own input. The problem is that there are $r+1$ processes that are not the winner of the Q_r object and may need to learn the identity of the winner. However, the Q_r object can only return the identity of the winner to at most r processes. To resolve this, each process that loses its COMPETE operation first overwrites its announced value with $\perp$ before calling QUERY on the Q_r object. If its QUERY operation is successful, then, as before, it reads and decides the announced value of the winner. However, if its QUERY operation was unsuccessful, then $r+1$ processes have already called QUERY on the Q_r object by this point. Prior to calling QUERY, these processes overwrote their announced values. Thus, exactly one value remains in the announcement array: the value of the winner. The code is provided in Algorithm 1.

▶ **Observation 1.** *Algorithm 1 solves the consensus task among $r+2$ processes.*

▶ **Lemma 2.** *The Q_r object has consensus number at least $r+2$.*

3.1 An implementation from $(r+2)$-consensus objects

We now give a wait-free implementation of a Q_r object shared among any finite number n of processes using only $(r+2)$-consensus objects and registers. In particular, this shows that the Q_r object has consensus number at most $r+2$.

Our implementation uses an array $cons[1 \dots n]$ of $(r+2)$-consensus objects, a register *gate*, and a fetch-and-increment object *count*. Note that fetch-and-increment objects have a wait-free implementation from 2-consensus objects (and, hence, from $(r+2)$-consensus objects, since $r \geq 0$) and registers among any finite number of processes [4].

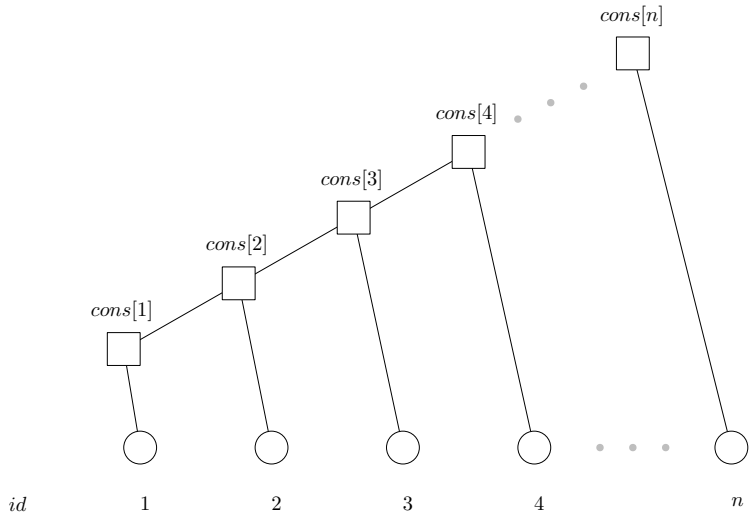


Figure 1 Tournament to determine winner.

Processes that call COMPETE participate in a tournament to decide which operation returns *true*. However, we must prevent the situation where, after a COMPETE operation returns $false$, another COMPETE operation begins and returns *true*. No linearization of such an execution would be consistent with the sequential specifications of COMPETE.

To prevent this issue, a process p_i performing COMPETE first checks the value of the shared register *gate*. If $gate \neq \perp$, the COMPETE operation returns $false$. Otherwise, p_i writes its own *id*, i, to *gate*. This ensures all processes calling COMPETE that write their *id* to *gate* are concurrent. This idea was used in the wait-free implementation of an n-process test-and-set object from 2-process test-and-set objects and registers [3].

After process p_i writes to *gate*, it continues by entering a tournament. It proposes i to $cons[i]$ through $cons[n]$ in order. If it receives a response other than i from one of these consensus objects, it immediately returns false and does not propose i to any further consensus objects. If the responses it receives from $cons[i]$ through $cons[n]$ are all i, then p_i returns *true*. The tournament is depicted in Figure 1. Note that $cons[1]$ is only used for simplifying the code and proof of correctness. It is otherwise unnecessary.

To perform QUERY, a process p_i first reads the value i' in *gate*. If $i' = \perp$, then no process performing COMPETE has written its *id* to *gate* and taken part in the tournament, so p_i returns $\perp$. If $i' \neq \perp$, p_i performs F&I on *count* and, if the value returned is greater than r, returns $\perp$. Note that at most r QUERY operations proceed past this point. If a COMPETE operation has already returned *true*, p_i could just propose to $cons[n]$ to get the *id* of the winner. However, this is not guaranteed. The competing processes may be slow or the winning process may have crashed before proposing to $cons[n]$. Thus, p_i attempts to assist participants in the tournament by propagating their *id*s. It begins by proposing i', the *id* it read from *gate*, to $cons[i']$. Then it proposes the decision of $cons[j-1]$ to $cons[j]$ for $i' < j \leq n$ in order. Finally, it returns the decision of $cons[n]$, which is the *id* of the winner.

The code for COMPETE and QUERY is presented in Algorithm 2.

We now prove the correctness of the implementation of a Q_r object given in Algorithm 2. Let $\mathcal{E}$ be any execution of the implementation.

We will say that a COMPETE or QUERY *operation* performs a step σ in the execution if the process performing the operation executes σ during the operation.

Algorithm 2 An implementation of Q_r.

Shared variables:
$cons[1 \dots n]$ is an array of $(r+2)$-consensus objects
gate is a register initialized to $\perp$
count is a fetch-and-increment object initialized to 1

```
c1:  function COMPETE( ) by process p_i
c2:      if gate.READ() ≠ ⊥ then
c3:          return false
c4:      end if
c5:      gate.WRITE(i)
c6:      for j ← i to n do
c7:          if cons[j].PROPOSE(i) ≠ i then
c8:              return false
c9:          end if
c10:     end for
c11:     return true
c12: end function
```

```
q1:  function QUERY( )
q2:      i' ← gate.READ()
q3:      if i' = ⊥ then
q4:          return ⊥
q5:      end if
q6:      if count.F&I() > r then
q7:          return ⊥
q8:      end if
q9:      start ← i'
q10:     for j ← start to n do
q11:         i' ← cons[j].PROPOSE(i')
q12:     end for
q13:     return i'
q14: end function
```

We consider three cases.

In the first case, suppose that no operation in execution $\mathcal{E}$ ever writes to *gate*. That is, suppose no COMPETE operation executes line c5 during execution $\mathcal{E}$. Then no COMPETE operation returns, since the value of *gate* remains $\perp$ throughout execution $\mathcal{E}$. Thus, line c3 is not executed, nor are lines c8 or c11. All COMPETE operations are therefore incomplete and we do not linearize any of them. All QUERY operations that complete return $\perp$ on line q4. We can define the linearization point of each of these QUERY operations to be the time at which the operation executes line q2. Since all linearized QUERY operations return $\perp$ and no COMPETE operation has been linearized, this linearization is consistent with the sequential specifications of a Q_r object.

In the second case, suppose that line c5 is executed by some operation in execution $\mathcal{E}$, but no value is ever proposed to any consensus object. That is, neither line c7 nor line q11 is executed by any operation in $\mathcal{E}$. Then let C be the COMPETE operation during which the first write to *gate* takes place and let p_w be the process that performs operation C. Define the linearization point of operation C to be the step at which it writes to *gate* in line c5. In this execution, every COMPETE operation that completes returns $false$ on line c3. For every such completed COMPETE operation, define its linearization point to be the step at which it reads from *gate* in line c2. It follows that for every completed COMPETE operation C', operation C is linearized before operation C', since the read operation performed by C' in line c2 returned a value other than $\perp$, meaning that the first write to *gate*, in line c5 of C, had already taken place.

We also linearize the QUERY operations as follows: Every QUERY operation that returns $\perp$ on line q4 is linearized at the point it executes the read of *gate* in line q2. Thus, every such operation Q is linearized before C, since Q reads $\perp$ from *gate*. The QUERY operations that perform F&I on *count* are linearized at the point they execute line q6. Each of these QUERY operations is linearized after C, since it reads a value other than $\perp$ from *gate* in line q2. Moreover, if one of these QUERY operations Q returns $\perp$, it does so after *count* has

already been incremented r times by r other QUERY operations that executed line q6. So there at least r QUERY operations linearized after C and before every such operation Q. This linearization is consistent with the sequential specifications of a Q_r object.

Finally, we consider any execution $\mathcal{E}$ where some value is proposed to a consensus object on line c7 or q11 during some operation in $\mathcal{E}$. Processes are referred to as *competitors* while they execute lines c5–c11 and as *queriers* while they execute lines q9–q13. Note that only competitors and queriers access consensus objects, and there are at most r queriers.

▶ **Observation 3.** *A process can only be a competitor in its first invocation of* COMPETE*, so every competitor has a distinct id.*

We say that i *is accepted by* a consensus object if it is the decision of the consensus object. If i is proposed, but is not accepted, we say it *is rejected from* the consensus object.

Competitor p_i only proposes its *id*, i, to $cons[j]$ if $j = i$ or if i was accepted by $cons[j-1]$. Every querier begins by reading some competitor's *id*, i', in *gate*. It proposes this value i' to $cons[i']$ in the first iteration of the for loop in line q10. In every subsequent iteration, it proposes the value accepted by $cons[j-1]$ to $cons[j]$. This gives us the following observation:

▶ **Observation 4.** *For any $j > 1$, the only possible values proposed to $cons[j]$ are j and the value accepted by $cons[j-1]$. The only possible value proposed to $cons[1]$ is 1.*

From Observation 4, we get the following useful results:

▶ **Lemma 5.** *If $i < j$ is proposed to $cons[j]$, then i was accepted by $cons[\ell]$, for all $i \leq \ell < j$.*

Proof. Suppose not. Consider the largest $\ell < j$ such that i was not accepted by $cons[\ell]$. If $\ell < j-1$, this means that i was accepted by $cons[\ell+1]$ and, hence, from the semantics of PROPOSE, was proposed to $cons[\ell+1]$. If $\ell = j-1$, then i was proposed to $cons[\ell+1]$ by assumption. In either case, since $i < l+1$, it follows from Observation 4 that the value i was accepted by $cons[\ell]$. This is a contradiction. ◀

▶ **Lemma 6.** *No process that proposes to the $(r+2)$-consensus object $cons[j]$ receives a response of $\perp$.*

Proof. By Observation 4, there are at most two distinct values proposed to $cons[j]$. Since every competitor only proposes its own *id*, by Observation 3, we conclude that there are at most 2 competitors that propose to $cons[j]$. No competitor proposes to $cons[j]$ more than once. Additionally, there are at most r queriers and each querier proposes to $cons[j]$ at most once. It follows that there are at most $r+2$ PROPOSE operations performed on $cons[j]$ and, hence, none of them receive a response of $\perp$. ◀

By examining the code, we can learn the following fact about the values proposed to consensus objects:

▶ **Observation 7.** *The value i is only proposed to $cons[j]$ if there exists a competitor with id i that has already written i to gate on line c5.*

We will now show that execution $\mathcal{E}$ is linearizable.

Recall that during execution $\mathcal{E}$, some value is proposed to some consensus object. Consider the smallest j such that some value is proposed to $cons[j]$. By Observation 4, the only value proposed to $cons[j]$ is j. It follows that j is accepted by $cons[j]$. Let w be the greatest value such that w is accepted by $cons[w]$ in execution $\mathcal{E}$. Let C_w be the first COMPETE operation of process p_w. By Observations 3 and 7, C_w exists and writes w to *gate* in line c5 (after

reading $\perp$ from *gate* in line c2). Thus, operation C_w is concurrent with the first write to *gate* and we linearize it at that point. Every COMPETE operation that returns on line c3 is linearized at the point it executes line c2 and every COMPETE operation that returns on line c8 is linearized at the point of its preceding proposal to a consensus object in line c7.

We now show that C_w is the only COMPETE operation that can return on line c11. This means that we have linearized every completed COMPETE operation.

▶ **Lemma 8.** *If i is accepted by $cons[n]$, then $i = w$.*

Proof. Since i is accepted by $cons[n]$, then, by Lemma 5, it is also accepted by $cons[\ell]$, for all $i \leq \ell < n$. Thus i was accepted by $cons[i]$ and, for any $j > i$, j was not accepted by $cons[j]$. So $i = w$. ◀

▶ **Corollary 9.** *The only* COMPETE *operation that can return true is operation C_w.*

Proof. If COMPETE operation C by process p_i returns *true*, then i is accepted at $cons[n]$ so, by Lemma 8, $i = w$. By Observation 3, only the first COMPETE operation of any process can return *true*, so $C = C_w$. ◀

▶ **Lemma 10.** *Any linearized* COMPETE *operation $C \neq C_w$ is linearized after C_w.*

Proof. Consider any such operation C. Since $C \neq C_w$ is linearized, C has completed and so it returns on line c3 or line c8 by Corollary 9. If operation C returns on line c3, it is linearized when it reads some value other than $\perp$ from *gate* in line c2. Since C_w is linearized at the earliest write to *gate* in execution $\mathcal{E}$, C_w is linearized before C. On the other hand, if operation C returns on line c8, then it performs a write to *gate* in line c5, which is at or after the linearization point of C_w. Operation C is linearized later, when it performs its last proposal to a consensus object in line c7. Thus, C_w is linearized before C. ◀

It follows that p_w is the winner in the linearization of execution $\mathcal{E}$. From Corollary 9, we know that C_w is the only operation that can return *true*. We must also argue that it does not return $false$:

▶ **Lemma 11.** *Operation C_w does not return $false$.*

Proof. For COMPETE operation C_w to return $false$, w must be rejected from $cons[j]$, for some $j > w$. Consider any such j. By Observation 4, w was accepted by $cons[j-1]$ and j was the only other value proposed to $cons[j]$. Since w was rejected from $cons[j]$, it follows that j was accepted by $cons[j]$. But this contradicts the definition of w. ◀

Finally, we can show that it is possible to linearize the QUERY operations. As in the second case, every QUERY operation that returns on line q4 is linearized at the point it reads $\perp$ from *gate* in line q2. This is before the first write to gate, which is when C_w is linearized. Every QUERY operation that calls F&I on *count* in line q6 is linearized at the point it executes line q6, which occurs after it reads a value other than $\perp$ from *gate* in line q2 and, thus, after the linearization point of C_w. Note that all completed QUERY operations either return on q4 or execute line q6. By the semantic of F&I, the QUERY operations that execute line q6 are linearized in increasing order of the values they receive from *count*. In particular, if some such QUERY operation receives a value greater than r, at least r other QUERY operations have been linearized before it (and after C_w).

▶ **Corollary 12.** *Every* QUERY *operation that receives a value of r or less from count can only return the id of the winner p_w.*

Proof. Any such QUERY operation can only return in line q13 and it returns the value accepted by $cons[n]$, which, by Lemma 8, is w. ◀

Thus, the linearization of execution $\mathcal{E}$ is consistent with the sequential specifications of a Q_r object. Since $\mathcal{E}$ was chosen arbitrarily, every execution of Algorithm 2 is linearizable. Note that the implementations of COMPETE and QUERY in Algorithm 2 are wait-free. Therefore, we have shown the following theorem:

▶ **Theorem 13.** *For $r \geq 0$, Q_r has consensus number $r+2$ and can be implemented in a wait-free manner from $(r+2)$-consensus objects and registers in a system with any finite number of processes.*

4 Implementing $O_{m,k}$ from $(m+1)$-consensus objects

In this section, we present the formal definition of the $O_{m,k}$ object from [1], followed by a description of a natural, but incorrect, approach for implementing the $O_{m,k}$ object from $(m+1)$-consensus objects and registers for any number of processes. We then give our wait-free implementation and prove that it is correct.

4.1 Sequential Specifications of $O_{m,k}$

The $O_{m,k}$ object, for $m, k \geq 2$, has a single operation SUGGEST, which takes a non-negative argument, and has the following specifications, where, for $j \in \{1, \ldots, k\}$, a_j is the argument of the $(j-1)m+1^{st}$ SUGGEST operation:

- For $j \in \{1, \ldots, k\}$, the $(j-1)m+1^{st}$ through jm^{th} SUGGEST operations return a_j.
- For $j \in \{1, \ldots, k-1\}$, the $km+j^{th}$ SUGGEST operation returns a_{k-j}.
- For $j > km+k-1$, the j^{th} SUGGEST operation returns $\perp$.

The first km SUGGEST operations performed on an $O_{m,k}$ object are called *prefix operations* and the next $k-1$ SUGGEST operations are called *suffix operations*. A more intuitive way to visualize the behaviour of SUGGEST is with the string $S_{m,k} = A_1^m A_2^m \ldots A_k^m A_{k-1} A_{k-2} \ldots A_1$. For $1 \leq j \leq km+k-1$, if A_g is the j^{th} character in $S_{m,k}$, then the j^{th} SUGGEST operation returns a_g, and we say it *belongs to group g*. For $j > km+k-1$, the j^{th} SUGGEST operation returns $\perp$.

4.2 An incorrect approach

A simple algorithm, due to Faith Ellen, for implementing $O_{m,2}$ from $(m+1)$-consensus objects and registers for any number of processes is as follows:

- Each time a process p_i performs SUGGEST(v), it first accesses a fetch-and-increment object to receive a distinct value x.
- If $x > 2m+1$, then p_i returns $\perp$.
- If $1 \leq x \leq m$, then p_i proposes v to an $(m+1)$-consensus object C_1. It writes the response to a shared register R_1 before returning it.
- If $m+1 \leq x \leq 2m$, then p_i proposes v to an $(m+1)$-consensus object C_2. It writes the response to a shared register R_2 before returning it.
- If $x = 2m+1$, p_i reads R_1 and then R_2. It returns the first non-$\perp$ value it reads. If both R_1 and R_2 are $\perp$, this means that all processes that received values at most $2m+1$ from the fetch-and-increment object are concurrent. In this case, p_i proposes v to C_1, and returns the response.

This algorithm correctly implements an $O_{m,2}$ object from $(m+1)$-consensus objects registers. For each of the consensus objects C_1 and C_2, the SUGGEST operation that first proposes a value to the consensus object can be linearized before the rest of the SUGGEST operations that propose to it.

However, this approach does not scale to $k > 2$. Consider the natural extension of this algorithm to $O_{m,3}$:

- Once again, each time a process p_i performs SUGGEST(v), it first accesses a fetch-and-increment object to receive a distinct value x.
- If $x > 3m + 2$, then p_i returns $\perp$.
- If $(j-1)m + 1 \leq x \leq jm$ for $j \in \{1, 2, 3\}$, then p_i proposes v to an $(m+1)$-consensus object C_j. It writes the response to a shared register R_j before returning it.
- If $x \in \{3m + 1, 3m + 2\}$, then p_i will propose to the consensus object associated with some group and return the response.

The problem with the above approach arises from the operations that receive $3m+1$ and $3m+2$ from the fetch-and-increment object. Firstly, these operations must propose to the consensus objects associated with different groups. This can be achieved as in [1] by using test-and-set objects. The more difficult issue is ensuring linearizability:

Consider an execution in which some process p_i receives $3m + 1$ from the fetch-and-increment object, reads $R_1 = R_2 = R_3 = \perp$, proposes v to some consensus object C_j, and returns the decision. Next, the m operations that received $(j-1)m+1, \ldots, jm$ from the fetch-and-increment object complete their operations, writing to R_j and returning the decision of C_j. Next, another process p_ℓ begins its SUGGEST operation with an argument that is different from the arguments of all preceding SUGGEST operations and receives $3m+2$ from the fetch-and-increment object. Note that since an entire group (of $m+1$ SUGGEST operations) has returned before p_ℓ began, p_ℓ must be performing a suffix operation when this execution is linearized. In particular, it cannot return the argument of its own SUGGEST operation.

If the other prefix operations have not made any progress (i.e. they have not taken any steps since receiving a value from the fetch-and-increment object), then p_ℓ cannot determine any of their arguments, unless it waits for them, which it cannot do. Even if each SUGGEST operation announces its argument before it accesses the fetch-and-increment object, p_ℓ does not know which prefix operations belong to the same group as p_i's operation and, so, cannot determine a value to return.

4.3 A wait-free implementation

The following implementation starts by assigning m prefix operations to each of the k groups, and then assigns the $k-1$ suffix operations to the first $k-1$ groups. It ensures that suffix operations can determine and propose the argument of some prefix operation in their group. To do this, we require a stronger synchronization mechanism than a fetch-and-increment object: namely, the Q_1 object.

To implement the $O_{m,k}$ object in a system with any finite number of processes, n, from $(m+1)$-consensus objects and registers, we use an array $cons[1 \ldots k]$ of $(m+1)$-consensus objects and an array $position[1 \ldots km]$ of Q_1 objects. Note that since the Q_1 object can be implemented by 3-consensus objects and registers for any finite number of processes and $m + 1 \geq 3$, $(m+1)$-consensus objects and registers can also implement the Q_1 object for any finite number of processes.

A process p_i performing SUGGEST(v) will perform COMPETE on objects *position*[1] through *position*[km] in order, until it wins one of them or loses all km of them.

If p_i is the winner of the Q_1 object *position*[j], it is performing a prefix operation. It will propose v to the consensus object *cons*[$\lceil \frac{j}{m} \rceil$] associated with its group and return its response.

If p_i fails to win any Q_1 object, it is either a suffix operation or it returns $\perp$. It accesses a fetch-and-increment object *count* that is initially 1. Note that fetch-and-increment objects have a wait-free implementation for any finite number of processes from 2-consensus objects and registers and, consequently, since $m + 1 > 2$, also from $(m + 1)$-consensus objects and registers. Let x be the response from *count* that p_i receives. If $x > k - 1$, p_i returns $\perp$. Otherwise, it is performing the suffix operation associated with group $k - x$. It performs QUERY on *position*[$(k - x)m$] to get the identity i' of some process that performed a prefix operation associated with group $k-x$. If each process announces the argument of its SUGGEST operation at the beginning of the operation, then p_i could read the value announced by $p_{i'}$, propose this value to *cons*[$k - x$], the consensus object associated with its group, and return the response. This ensures that the arguments of prefix operations are the only non-$\perp$ values returned.

There is a slight problem with this approach. Process $p_{i'}$ could have performed another SUGGEST operation afterwards that overwrote the value it announced before winning *position*[$(k - x)m$]. Instead, we use an array $A_j[1 \ldots n]$ of n registers associated with each *position*[j]. Before performing a COMPETE operation on *position*[j], each process p_ℓ writes the argument of its current SUGGEST operation to $A_j[\ell]$, provided it has not previously written there. Then process p_i can read $A_{(k-x)m}[i']$ to find the argument of a prefix operation in group $k - x$.

The code for the implementation is given in Algorithm 3.

4.4 Correctness

We will show that the implementation of an $O_{m,k}$ object given in Algorithm 3 is linearizable. Consider any execution $\mathcal{E}$ of this implementation.

We will say that a SUGGEST *operation* performs a step σ in the execution if the process performing the operation executes σ during the operation.

A SUGGEST operation S *fills* the Q_1 object *position*[j] if it performs the COMPETE operation on *position*[j] that returns *true*. In this case, operation S will be ordered as one of the m prefix operations belonging to group $\lceil \frac{j}{m} \rceil$.

If a SUGGEST operation S performs the F&I on line s10, receiving $x < k$, then S will be ordered as the suffix operation belonging to group $k - x$.

▶ **Lemma 14.** *When a* SUGGEST *operation completes iteration j of the for loop on line s2, position*[j] *is filled.*

Proof. Let S be a SUGGEST operation by process p_i that completes iteration j. If, during SUGGEST operation S, p_i reads $A_j[i] = \perp$, then it performs the COMPETE operation on *position*[j] in line s5 before iteration j completes. If, on the other hand, p_i reads $A_j[i] \neq \perp$ on line s3, then there must have been a SUGGEST operation S' by process p_i that completed before p_i began operation S, in which the COMPETE operation on *position*[j] in line s5 was executed. In either case, by the time that S completes iteration j, at least one COMPETE operation has been performed on *position*[j]. The first of these COMPETE operations must have returned *true*, so *position*[j] is filled. ◀

Algorithm 3 An implementation of $O_{m,k}$.

Shared variables:
$cons[1 \dots k]$ is an array of $(m+1)$-consensus objects
$A_j[1 \dots n]$ is an array of registers initialized to $\perp$, for $j \in \{1, \dots, km\}$
$position[1 \dots km]$ is an array of Q_1 objects
$count$ is a fetch-and-increment object initialized to 1

```
s1:  function SUGGEST(v) by process p_i
s2:      for j ← 1 to km do
s3:          if A_j[i].READ() = ⊥ then
s4:              A_j[i].WRITE(v)
s5:              if position[j].COMPETE() then
s6:                  return cons[⌈j/m⌉].PROPOSE(v)
s7:              end if
s8:          end if
s9:      end for
s10:     x ← count.F&I()
s11:     if x > k − 1 then
s12:         return ⊥
s13:     end if
s14:     group ← k − x
s15:     member ← position[group × m].QUERY()
s16:     value ← A_{group×m}[member]
s17:     return cons[group].PROPOSE(value)
s18: end function
```

From Lemma 14, we get the following important corollaries:

▶ **Corollary 15.** *All* SUGGEST *operations that perform line s10 can be ordered after all* SUGGEST *operations that fill some position.*

Proof. A SUGGEST operation S that performs line s10 must first perform all km iterations of the for loop on line s2. It follows from Lemma 14 that there is no SUGGEST operation that fills a position and starts after S completes. ◀

▶ **Corollary 16.** *For $j < j'$, a* SUGGEST *operation that fills $position[j]$ can be ordered before a* SUGGEST *operation that fills $position[j']$.*

Proof. Consider a SUGGEST operation S that fills $position[j]$ and a SUGGEST operation S' that fills $position[j']$, with $j < j'$. To fill $position[j']$, S' must have first completed iteration j of the for loop in line s2. Thus, by Lemma 14, $position[j]$ was already filled before S' completed. By assumption, $position[j]$ was filled by operation S. Hence, S can be linearized before S'. ◀

▶ **Observation 17.** *If the* F&I *operation performed on line s10 by* SUGGEST *operation S returns x, and the* F&I *operation performed on line s10 by* SUGGEST *operation S' returns $x' > x$, then S can be ordered before S'.*

We are now able to order the SUGGEST operations in execution $\mathcal{E}$. In particular, by Corollary 16, the SUGGEST operations that fill $position[\ell]$, for $(j-1)m+1 \leq \ell \leq jm$, can be

ordered as the m prefix operations of group j. Moreover, by Corollary 15 and Observation 17, the SUGGEST operation that receives $x < k$ from the F&I on line s10 can be ordered as the suffix operation of group $k - x$ and any SUGGEST operation that receives $x > k - 1$ from the F&I on line s10 can be ordered after the first $km + k - 1$ operations.

All that remains is to order the prefix operations belonging to each group with respect to each other and show that the results of the operations in this ordering are consistent with the specifications of $O_{m,k}$. We first make the following two observations:

▶ **Observation 18.** *If any* SUGGEST *operation that belongs to group g returns a value, it returns the decision of $cons[g]$.*

▶ **Observation 19.** *If the suffix operation belonging to group g proposes to $cons[g]$, it proposes the value of the* SUGGEST *operation that fills $position[gm]$.*

In execution $\mathcal{E}$, if no SUGGEST operation belonging to group g proposed to $cons[g]$, then, by Corollary 16, we can order the prefix operations in order of the indices of the Q_1 objects they fill.

Otherwise, one or more values were proposed to $cons[g]$. By Observation 19, only the arguments of the prefix operations belonging to group g are proposed to $cons[g]$. Let v be the value first proposed to (and, hence, decided by) $cons[g]$ and let j be the smallest index, with $(g-1)m + 1 \leq j \leq gm$, such that the prefix operation S that fills $position[j]$ has argument v. No prefix operation in group g that fills $position[j']$, for $j' < j$, finished before S began, since v is the value first proposed to $cons[g]$. It follows that we can order every other prefix operation belonging to group g after S, in order of the indices of the Q_1 objects they fill.

By Observations 18 and 19, the results of the SUGGEST operations in this ordering are consistent with the sequential specifications of the $O_{m,k}$ object. Note that the implementation of SUGGEST in Algorithm 3 is wait-free. Thus, we have shown the following theorem:

▶ **Theorem 20.** *There is a wait-free implementation of the $O_{m,k}$ object from $(m+1)$-consensus objects and registers for any $m \geq 2$ and any $k \geq 2$.*

5 Conclusions

In this paper, we introduced the Q_r object for $r \geq 0$, showed that it has consensus number $r + 2$, and presented a wait-free implementation of Q_r from $(r + 2)$-consensus objects and registers in a system with any finite number of processes. Using this object, we showed that, for any $m, k \geq 2$, there is a wait-free implementation of $O_{m,k}$ from $(m + 1)$-consensus objects and registers in a system with any finite number of processes. This means that there is an infinite sequence $O_{m,2}, O_{m,3}, \ldots$ of objects with increasing computational power, all of which have consensus number m and are computationally less powerful than the $(m+1)$-consensus object. From Rachman's result, we know that for every $m' > m \geq 1$, there exists a nondeterministic object X with consensus number m which cannot be implemented using m'-consensus objects and registers in some system with a finite number of processes. If the same property is true when restricted to deterministic objects, our implementation of $O_{m,k}$ from $(m + 1)$-consensus objects and registers shows that $O_{m,k}$ objects cannot be used to prove this.

The $O_{m,k}$ objects, for $k \geq 2$, are the only known deterministic objects of consensus number m that cannot be implemented in a wait-free manner by m-consensus objects and registers for some finite number of processes. It may be the case that any deterministic object with consensus number m has a wait-free implementation from $(m + 1)$-consensus

objects in a system with any finite number of processes. One approach to proving this is to show that, for every object O with consensus number m, there exists $k \geq 2$ such that O is computationally less powerful than $O_{m,k}$. Another interesting question is whether there exists some deterministic object with consensus number m that can implement any other deterministic object with consensus number m in a system with any finite number of processes.

Acknowledgements. I would like to thank my supervisor, Faith Ellen, for her constant encouragement and invaluable guidance and my anonymous reviewers for their very helpful feedback. The implementation of $(r + 2)$-consensus from Q_r objects and registers given in Algorithm 1 is due to Leqi Zhu, who simplified my original implementation.

References

1 Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 97–106, 2016.

2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 218–227, 2006.

3 Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, pages 85–94, 1992.

4 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 159–170, 1993.

5 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

6 Ophir Rachman. Anomalies in the wait-free hierarchy. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, pages 156–163, 1994.

RADON: Repairable Atomic Data Object in Networks*

Kishori M. Konwar[1], N. Prakash[2], Nancy A. Lynch[3], and Muriel Médard[4]

1 **Department of EECS, MIT, Cambridge, MA, USA**
`kishori@csail.mit.edu`
2 **Department of EECS, MIT, Cambridge, MA USA**
`prakashn@mit.edu`
3 **Department of EECS, MIT, Cambridge, MA, USA**
`lynch@csail.mit.edu`
4 **Department of EECS, MIT, Cambridge, MA, USA**
`medard@mit.edu`

Abstract

Erasure codes offer an efficient way to decrease storage and communication costs while implementing atomic memory service in asynchronous distributed storage systems. In this paper, we provide erasure-code-based algorithms having the additional ability to perform background repair of crashed nodes. A repair operation of a node in the crashed state is triggered externally, and is carried out by the concerned node via message exchanges with other active nodes in the system. Upon completion of repair, the node re-enters active state, and resumes participation in ongoing and future read, write, and repair operations. To guarantee liveness and atomicity simultaneously, existing works assume either the presence of nodes with stable storage, or presence of nodes that never crash during the execution. We demand neither of these; instead we consider a natural, yet practical network stability condition $N1$ that only restricts the number of nodes in the crashed/repair state during broadcast of any message.

We present an erasure-code based algorithm $RADON_C$ that is always live, and guarantees atomicity as long as condition $N1$ holds. In situations when the number of concurrent writes is limited, $RADON_C$ has significantly improved storage and communication cost over a replication-based algorithm $RADON_R$, which also works under $N1$. We further show how a slightly stronger network stability condition $N2$ can be used to construct algorithms that never violate atomicity. The guarantee of atomicity comes at the expense of having an additional phase during the read and write operations.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Atomicity, repair, fault-tolerance, storage cost, erasure codes

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.28

1 Introduction

We consider the problem of designing algorithms for distributed storage systems (DSSs) that offer consistent access to stored data. Large scale DSSs are widely used by several industries, and also widely studied by academia for a variety of applications ranging from e-commerce

* The work is supported in part by AFOSR under grants FA9550-14-1-043, FA9550-14-1-0403, and in part by NSF under awards CCF-1217506, CCF-0939370.

to sequencing genomic-data. The most desirable form of consistency is atomicity, which in simple terms, gives the users of the data service the impression that the various concurrent read and write operations take place sequentially. Implementations of atomicity on an asynchronous system under message passing framework, in the presence of failures, is often challenging. Traditional implementations[4], [14] use replication of data as the mechanism of fault-tolerance; however they suffer from the problem of having high storage cost, and communication costs for read and write operations.

Erasure codes provide an efficient way to decrease storage and communication cost in atomicity implementations. An $[n, k]$ erasure code splits the value v, say of size 1 unit into k elements, each of size $\frac{1}{k}$ units, creates n *coded elements*, and stores one coded element per server. The size of each coded element is also $\frac{1}{k}$ units. A class of erasure codes known as Maximum Distance Separable (MDS) codes have the property that value v can be reconstructed from any k out of these n coded elements. While it is known that usage of erasure codes in asynchronous decentralized storage systems do not offer all the advantages as in synchronous centralized systems [29], erasure code based algorithms like in [1], [13], [8], or [19] for implementing consistent memory service offer significant storage and communication cost savings over replication based algorithms, in many regimes of operation. For instance CASGC [8] improves the costs under the scenario when the number of writes concurrent with a read is known to be limited, whereas SODA [19] trades-off write cost in order to optimize storage cost, which is meaningful in systems with infrequent writes. Both CASGC and SODA are based on MDS codes.

In this work, we consider the additional important issue of repairing crashed nodes without disrupting the storage service. Failure of storage nodes is a norm rather than an exception in large scale DSSs today, primarily because of the usage of commodity hardware for affordability and scalability reasons. Replication based algorithms in [4], [14] and erasure-code based algorithms in [1], [8], or [19] do not consider repair of crashed nodes; instead assume that a crashed node remains so for the rest of the execution. Algorithms in [13], [15] consider background repair of crashed nodes; however they assume either the presence of nodes having stable storage, whose content is unaffected by crashes, or presence of a subset of nodes that never crash during the entire execution. We relax both these assumptions in this work. In our model, any one of the storage nodes can crash; further, we assume that a crashed node loses all its data, both volatile as well as stable storage. A repair operation of a node in the crashed state is triggered externally, and is carried out by the concerned node via message exchanges with other active nodes in the system. Upon completion of repair, the node (with the same id) re-enters active state, and resumes participation in ongoing and future read, write, and repair operations.

It is natural to expect a restriction on the number of crash and repair operations in relation to the read and write operations; the authors of [15] show an impossibility result in this direction, for guaranteeing liveness and atomicity, simultaneously. We formulate network stability conditions $N1$ and $N2$, which can be used to limit the number of crash and repairs operations overlapping with a client operation. These conditions are algorithm independent, and most likely to be satisfied in any practical storage network. At a high level, the condition $N1$ restricts the set of servers that can be in the crashed or repair state any time a process (client or server) *pings* all the n servers with corresponding messages. Condition $N2$ is slightly stronger than $N1$, and restricts the set of servers that can be in the crashed or repair state if the process wants to *ping-pong* a fraction of the servers. In a ping-pong, it is expected that the servers which receive a message also respond back to the sender of the message.

1.1 Summary of Our Contributions

We first present an impossibility result for an asynchronous DSS allowing background repair of crashed nodes, where there is no restriction on the number of crash and repair operations that occur during a client operation. We show that it is impossible to simultaneously achieve liveness and atomicity in such a system, even if all the crash and repair operations occur sequentially during the execution (i.e., at most one node remains in the crash or repair state at any point during the execution).

We then consider the problem of erasure-code based algorithm design under the network stability condition $N1$. We present the algorithm in two stages. First we present a replication-based algorithm $RADON_R$, which performs background-repair, and guarantees atomicity and liveness of operations under $N1$, if more than $3/4^{\text{th}}$ of all servers remain active during any ping operation. The write and read phases are almost identical to those of the ABD algorithm [4], except that during a write we expect responses from more than $3/4^{\text{th}}$ of all the servers, while in ABD responses are expected only from a majority of servers. A repair operation in $RADON_R$ is simply a read operation initiated by the concerned server. Thus the algorithm itself is simple; however, the proof of atomicity gets complicated because of the fact that a repair operation can potentially restore the contents of a node to a version that is older than what was present before the crash. We show how the network stability condition can be used to prove atomicity, and this proof is the key takeaway from $RADON_R$ towards constructing the erasure-code based algorithm.

Our erasure-code based algorithm $RADON_C$ uses $[n, k]$ MDS codes, and is a natural adaptation of $RADON_R$ for the usage of codes. A key challenge while using erasure codes is ensuring liveness of read operations, in the presence of concurrent write operations. Various techniques are known in literature to handle this challenge; for instance, [13] assumes synchronous write phases, [8] limits the number of writes concurrent with a read, while [19] uses an $O(n^2)$ write protocol to guarantee liveness of reads. In this work, like in [8], we make the assumption that the number of write operations concurrent with any read operation is limited by a parameter δ, which is known a priori. However, the usage of the concurrency bound differs from that of the CASGC algorithm in [8]; for instance, CASGC has three rounds for write operations, while $RADON_C$ uses only two rounds. In $RADON_C$, each server maintains a list of up to $\delta + 1$ coded elements, corresponding to the latest $\delta + 1$ versions received as a result of the various write operations. In comparison with $RADON_R$ where a writer expects responses from more than $3/4^{\text{th}}$ of all servers, a write operation in $RADON_C$ expects responses from more than $\frac{3n+k}{4}$ servers. During a read operation, the client reads the lists from more than $\frac{n+k}{2}$ nodes before decoding the value v. Like in $RADON_R$, a repair operation in $RADON_C$ is essentially a read operation by the concerned node; however this time the concerned node creates a list (instead of just one version) by decoding as many possibles versions that it can from the $\lceil \frac{n+k}{2} \rceil$ responses. Liveness and atomicity of operations are proved under network stability condition $N1$, if more than $\frac{3n+k}{4}$ servers remain active during any ping operation. $RADON_C$ has substantially improved storage and communication costs than $RADON_R$, when the concurrency bound δ is limited; see Table 1 for a comparison.

In both $RADON_R$ and $RADON_C$, violation of the network stability condition $N1$ can result in executions that are not atomic, which might not be preferable in certain applications. The choice of consistency over liveness, or vice versa, is the subject matter of a wide range of discussions and perspectives among system designers and software engineers. For example, BigTable, a DSS by Google, prefers safety over liveness [9], whereas, Amazon's Dynamo does not compromise liveness but settles for *eventual consistency* [10]. Our third algorithm

Table 1 Performance comparison of $RADON_R$, $RADON_C$ and $RADON_R^{(S)}$, where n is the number of servers, and δ is the maximum number of writes concurrent with a read or a repair operation. See Section 7 for a justification of the costs.

Algorithm	Write Cost	Read Cost	Storage Cost	Safe under	Live under
$RADON_R$	n	$2n$	n	$N1$	$N1$
$RADON_C$	$\frac{n}{k}$	$(\delta+2)\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$N1$	$N1$
$RADON_R^{(S)}$	n	$2n$	n	*always*	$N2$

$RADON_R^{(S)}$, which is replication-based, is designed to guarantee atomicity during every execution. Liveness is guaranteed under the slightly more stringent condition of $N2$, with more than $3/4^{\text{th}}$ of all servers remaining active during any ping-pong operation. The guarantee of atomicity of every execution also needs extra phases for read and write operations, when compared to $RADON_R$. The design of an erasure-coded version of $RADON_R^{(S)}$ that never violates atomicity, is an interesting direction that we leave out for future work.

1.2 Other Related Work

Dynamic Reconfiguration: Our setting is closely related to the problem of implementing a consistent memory object in a dynamic setting, where nodes are allowed to voluntarily leave and join the network. The problem involves dynamic reconfiguration of the set of nodes that take part in client operations, which is often implemented via a *reconfig* operation that is initiated by any of the participating processes, including the clients. Any node that wants to leave/join the network makes an announcement, via a *leave/join* operation, before doing so. The problem is extensively studied in the field of distributed algorithms [22], [3], [30], [6], [5]; review and tutorial articles appear in [2], [31], [24].

In our context, the problem of node repair could in fact be thought of as one of dynamic reconfiguration, wherein an involuntary crash is simulated by a voluntary leave operation without an explicit announcement. In this case, a new node joins as a replacement node via the *join* operation, which can be considered as the analogue of a *repair* operation. In the setting of dynamic reconfiguration, every node has a distinct identity; thus the replacement node joins the network with a new identity that is different from the identity of the crashed node [2]. This demands a reconfiguration of the set of participating nodes after every repair. Such reconfigurations get in the way of client operations, and add to the latency of read and write operations [24], in practical implementations. Clearly, a repair operation as considered in this work does not demand any reconfiguration, since a repaired node has the same identity as the crashed node. Also, the current work shows that modeling repair via a static system, permits design of algorithms where clients remain oblivious to the presence of repair operations. Furthermore, addressing storage and communication costs is not the focus of the works in dynamic reconfigurations; specifically, it is not known as to how erasure codes can be advantageously used in dynamic settings. Our $RADON_C$ algorithm shows that when repair is carried out under a static model, it is indeed possible to advantageously use erasure to reduce costs, when the number of concurrent writes are limited.

We make additional comparisons between our model and results to those found in works on dynamic reconfiguration. Several impossibility results exist in the context of implementing a dynamic atomic register and simultaneously guaranteeing liveness; the authors in [30] argue impossibility if there are infinitely many reconfigurations during an execution, while the authors in [6] argue an impossibility when there is no upper bound on message delay. We see, not surprisingly, that even in the problem of repair, we need to suitably limit the number

of crash and repair operations that occur in an execution, even if all crash and repairs are sequentially ordered. In [5], the authors implement a dynamic atomic register under a model that has an (unknown) upper bound D on any point-to-point message delay, and where the number of reconfigurations in any D units of time is limited. Our network condition $N1$ is similar, except that 1) we limit the number of crash and repairs during any broadcast messaging, instead of point-to-point messaging, and 2) we do not assume any bound on the message delay. In practice, limiting number of repairs during broadcast instead of every point-to-point messaging offers resiliency against *straggler* nodes, which refer to the nodes having the worst delays among all nodes. We would also like to note that the algorithm in [5] does not guarantee atomicity, if the number of reconfigurations in D units of time is higher than a set number. This appears similar to $RADON_R$, where atomicity is not guaranteed if we do not satisfy stability condition $N1$. While we show how the slightly tighter model $N2$ can be used to always guarantee atomicity, it is an interesting question as to whether the model $N2$ can be adopted in the work of [5] so as to always guarantee atomicity.

Repair-Efficient Erasure Codes for Distributed Storage: Recently, a large class of new erasure/network codes for storage have been proposed (see [11] for a survey), and also tested in networks [17], [27], [25], where the focus is efficient storage of immutable data, such as, archival data. These new codes are specifically designed to optimize performance metrics like repair-bandwidth and repair-time (of failed servers), and offer significant performance gains when compared to the traditional Reed-Solomon MDS codes [26]. It needs to be explored if these codes can be used in conjunction with the $RADON_C$ algorithm, to further improve the performance costs.

Other Works on using Erasure Codes: Applications of erasure codes to Byzantine fault tolerant DSSs are discussed in [7], [12], [16]. In [29], the authors consider algorithms that use erasure codes for emulating *regular* registers. Regularity [21], [28] is a weaker consistency notion than atomicity.

The rest of the document is organized as follows. Our system model appears in Section 2. The impossibility result, and the network stability conditions appear in Section 3. The three algorithms appear in Sections 4, 5 and 6, respectively. In Section 7, we discuss the storage and communication costs of the algorithms. Section 8 concludes the paper. Due to lack of space, detailed proofs are omitted here; these can be found in the extended version [20].

2 Models and definitions

Processes and Asynchrony: We consider a distributed system consisting of *asynchronous* processes, each with a unique identifier (ID), of three types: a set of *readers*, $\mathcal{R}$; a set of *writers*, $\mathcal{W}$; and a set of n *servers*, $\mathcal{S}$. The readers and writers are together referred to as clients. The set $\mathcal{R} \cup \mathcal{W} \cup \mathcal{S}$ forms a totally ordered set under some defined relation ($>$). The reader and writer processes initiate *read* and *write* operations respectively, and communicate with the servers using messages. A reader or writer can invoke a new operation only after all previous operations invoked by it has completed. The property is referred to as the *well-formedness* property of an execution. We assume that every client/server is connected to every other server via a reliable communication link; thus as long as the destination process is non-faulty, any message sent on the link eventually reaches the destination process.

Crash and Recovery: A client may fail at any point during the execution. At any point during the execution, a server can be in one (and only one) of the following three states: *active*, *crashed* or *repair*. A crash event triggers a server to enter the *crashed* state from an *active* state. The server remains in the *crashed* state for an arbitrary amount of time, but eventually is triggered by a repair event to enter the *repair* state. Crash and repair events are assumed to be externally triggered. A server in the *repair* state can experience another crash event, and go back to the *crashed* state. A server in the *crashed* state does not perform any local computation. The server also does not send or receive messages in the *crashed* state, i.e., any message reaching the server in a *crashed* state is lost. A server which enters the *repair* state has all its local state variables set to default values, i.e., a crash event causes the server to lose all its state variables. A server in the *repair* state can perform computations like in the *active* state.

Atomicity and Liveness: We aim to implement only one atomic read/write memory object, say x, under the MWMR setting on a set of servers, because any shared atomic memory can be emulated by composing individual atomic objects. The object value v comes from some set V; initially v is set to a distinguished value v_0 ($\in V$). Reader r requests a read operation on object x. Similarly, a write operation is requested by a writer w. Each operation at a non-faulty client begins with an *invocation step* and terminates with a *response step*. An operation is *incomplete* when its invocation step does not have the associated response step; otherwise it is *complete*.

By *liveness of a read or a write operation*, we mean that during any well-formed execution, any read or write operation respectively initiated by a non-faulty reader or writer completes, despite the crash failure of any other client. By *liveness of repair* associated with a crashed server, we mean that the server which enters a crashed state eventually re-enters the active state, unless it experiences a crash event during every repair operation that the server attempts. The liveness of repair holds despite the crash failure of any other client.

Background on Erasure coding: In $RADON_C$, we use an $[n, k]$ linear MDS code [18] over a finite field $\mathbb{F}_q$ to encode and store the value v among the n servers. An $[n, k]$ MDS code has the property that any k out of the n coded elements can be used to recover (decode) the value v. For encoding, v is divided[1] into k elements $v_1, v_2, \ldots v_k$ with each element having size $\frac{1}{k}$ (assuming size of v is 1). The encoder takes the k elements as input and produces n coded elements $c_1, c_2, \ldots, c_n$ as output, i.e., $[c_1, \ldots, c_n] = \Phi([v_1, \ldots, v_k])$, where Φ denotes the encoder. For ease of notation, we simply write $\Phi(v)$ to mean $[c_1, \ldots, c_n]$. The vector $[c_1, \ldots, c_n]$ is referred to as the codeword corresponding to the value v. Each coded element c_i also has size $\frac{1}{k}$. In our scheme we store one coded element per server. We use Φ_i to denote the projection of Φ on to the i^{th} output component, i.e., $c_i = \Phi_i(v)$. Without loss of generality, we associate the coded element c_i with server i, $1 \leq i \leq n$.

Storage and Communication Cost: We define the total storage cost as the size of the data stored across all servers, at any point during the execution of the algorithm. The communication cost associated with a read or write operation is the size of the total data that

[1] In practice v is a file, which is divided into many stripes based on the choice of the code, various stripes are individually encoded and stacked against each other. We omit details of representability of v by a sequence of symbols of $\mathbb{F}_q$, and the mechanism of data striping, since these are fairly standard in the coding theory literature.

gets transmitted in the messages sent as part of the operation. We assume that metadata, such as version number, process ID, etc. used by various operations is of negligible size, and is hence ignored in the calculation of storage and communication cost. Further, we normalize both the costs with respect to the size of the value v; in other words, we compute the costs under the assumption that v has size 1 unit.

3 Network Stability Conditions

3.1 An Impossibility Result

The crash and recovery model described in Section 2 does not impose any restriction on the *rate of crash events, and repair operations* that happen in the system. In other words, the model described above does not limit in any manner the number of crash events/repair operations, which can overlap with any a client operation. In [15], the authors showed that without such restrictions, it is impossible to implement a shared atomic memory service, which guarantees liveness of operations. Below, we state an impossibility result which holds even if there is at most one server in the crashed/repair state at any point during the execution. We then introduce network stability conditions that enable us impose restrictions on the number of crash/repair events that overlap with any operation.

▶ **Theorem 1.** *It is impossible to implement an atomic memory service that guarantees liveness of reads and writes, under the system model described in Section 2, even if* 1*) there is at most one server in the crashed/repair state at any point during the execution, and* 2*) every repair operation completes, and takes the repaired server back to the active state.*

3.2 Network Stability Conditions N1 and N2

We begin with the notions of a group-send operation, and effective consumption of a message.

Group-send operation: The group-send operation is used to abstract the operation of a process sending a list of n messages $\{m_1, \cdots, m_n\}$ to the set of all n servers $\{s_1, \ldots, s_n\} = \mathcal{S}$, where message m_i is send to server $s_i, 1 \leq i \leq n$. Note that this is a mere abstraction of the process sending out n point-to-point messages sequentially to n servers, without interleaving the "send" operations with any significant local computations or waiting for any external inputs. The operation is no more powerful then sending n consecutive messages. The operation is written as *group-send*$([m_1, m_2, \cdots, m_n])$. In the event $m_i = m, \forall i$, we simply write *group-send*(m). Our model allows the sender to fail while executing the *group-send* operation, in which case only a subset of the n servers receive their corresponding messages.

Effective Consumption: We say a process effectively consumes a message m, if it receives m, and executes all steps of the algorithm that depend only on the local state of the process, and the message m; in other words, the process executes all the steps that do not require any further external messages.

▶ **Definition 2** (Network Stability Conditions)**.** Consider a process p executing a *group-send* $([m_1, m_2, \cdots, m_n])$ operation, and consider the following statements:

(a) (i) There exists a subset $\mathcal{S}_\alpha \subseteq \mathcal{S}$ of $|\mathcal{S}_\alpha| = \lceil \alpha n \rceil$ servers, $0 < \alpha < 1$, all of which effectively consume their respective messages from the group-send operation, and (ii) all the servers in $\mathcal{S}_\alpha$ remain in the active state during the interval $[T_1\ T_2]$, where T_1 denotes the point of time of invocation of the *group-send* operation, and T_2 denotes the earliest

point of time in the execution at which all of the servers in $\mathcal{S}_\alpha$ complete the effective consumption of their respective messages.

(b) Further, if effective consumption of the message m_i by server s_i involves sending a response back to the process p, for all $s_i \in \mathcal{S}_\alpha$, then all servers in $\mathcal{S}_\alpha$ remain in the active state during the interval $[T_1\ T_3]$, where T_3 denotes the earliest point of time in the execution at which the process p completes effective consumption of the responses from the all the servers in $\mathcal{S}_\alpha$.

If the network satisfies Statement (a) for every execution of a group-send operation by any process, we say that it satisfies network stability condition $N1$ with parameter α. If the network satisfies Statements (a) and (b) for every execution of a group-send operation by any process, we say that it satisfies network stability condition $N2$ with parameter α.

Clearly, $N2$ implies $N1$. Note that the set $\mathcal{S}_\alpha$ which needs to satisfy the conditions need not be the same for various invocations of group-send operations by either the same or distinct processes. Also, note that in condition $N2$, the process p might crash before completing the effective consumption of the responses from the servers in $\mathcal{S}_\alpha$. In this case we only expect Statement (a) to be satisfied, and not Statement (b). Furthermore, in both $N1$ and $N2$, we do not expect any of these statements to be true, if process p crashes after partial execution of the group-send operation.

4 The $RADON_R$ Algorithm

In this section, we present the $RADON_R$ algorithm, and prove its liveness and atomicity properties for networks that satisfy the network condition $N1$ with $\alpha > \frac{3}{4}$. We begin with some useful notation. Tags are used for version control of the object values. A tag t is defined as a pair (z, w), where $z \in \mathbb{N}$ and $w \in \mathcal{W}$ denotes the ID of a writer. We use $\mathcal{T}$ to denote the set of all the possible tags. For any two tags $t_1, t_2 \in \mathcal{T}$, we say $t_2 > t_1$ if (i) $t_2.z > t_1.z$ or (ii) $t_2.z = t_1.z$ and $t_2.w > t_1.w$. Note that $(\mathcal{T}, >)$ is a totally ordered set.

The protocols for writer, reader, and servers are shown in Fig. 1. Each server stores two state variables (i) (t_{loc}, v_{loc}) – a tag and value pair, initially set to (t_0, v_0), (ii) *status* – a variable that can be in either *active* or *repair* state.

The write and read operations are very similar to those in the ABD algorithm [4], and each consists of two phases. In the first phase, *get-tag*, of a write operation π, the writer queries all servers for local tags, awaits responses from a majority of servers, and selects the maximum tag t^* from among the responses. Next, the writer executes the *put-data* phase, during which a new tag $t_w = tag(\pi)$ is created by incrementing the integer part of t^*, and by incorporating the writer's own ID. The writer then sends pair (t_w, v) to all servers, and awaits acknowledgments (acks) from $\lceil \frac{3n+1}{4} \rceil$ servers before completing the operation. The two phases are identical to those of the ABD algorithm [4], except for the fact that during the second phase, ABD expects acks from only a majority of servers, whereas here we need from $\lceil \frac{3n+1}{4} \rceil$ servers. During a read operation ρ, the reader in the *get-data* phase queries all the servers in S for the respective local tag and value pairs. Onces it receives responses from a majority of servers in S, it picks the pair with the highest tag, which we designate as $t_r = tag(\pi)$. In the subsequent *put-data* phase, the reader writes back the tag t_r and the corresponding value v_r to all servers, and terminates after receiving acknowledgments from $\lceil \frac{3n+1}{4} \rceil$ servers. Once again, we remark that both phases in the read are identical to those of the ABD algorithm, except for the difference in the number of the servers from which acks are expected in the second write-back phase. Note that, during both the write and operations, a server responds to an incoming message only if it is in the active state.

Fig. 1 The protocols for writer, reader, and any server $s \in \mathcal{S}$ in $RADON_R$.

write(v):
get-tag:
group-send(QUERY-TAG)
Await responses from majority
Select the max tag t^*
put-data:
$t_w = (t^*.z + 1, w)$
group-send((PUT-DATA, (t_w, v)))
Terminate after $\lceil \frac{3n+1}{4} \rceil$ acks.

read:
get-data:
group-send(QUERY-TAG-DATA)
Await responses from majority
Select (t_r, v_r), with max tag.

put-data :
group-send((PUT-DATA, (t_r, v_r)))
Wait for $\lceil \frac{3n+1}{4} \rceil$ acks
Return v_r

Server $s \in \mathcal{S}$:
State Variables:
$(t_{loc}, v_{loc}) \in \mathcal{T} \times \mathcal{V}$, initially (t_0, v_0)
$status \in \{active, repair\}$, initially $active$

get-tag-resp, **recv** QUERY-TAG **from writer** w:
if $status = active$ **then**
Send t_{loc} to w

get-data-resp, **recv** QUERY-TAG-DATA **from reader** r:
if $status = active$ **then**
Send (t_{loc}, v_{loc}) to r

put-data-resp, **recv** PUT-DATA, (t, v) **from client** c :
if $status = active$ **then**
if $t > t_{loc}$ **then**
$(t_{loc}, v_{loc}) \leftarrow (t, v)$
Send ack to c.

init-repair :
$status \leftarrow repair$
$(t_{loc}, v_{loc}) \leftarrow (t_0, v_0)$
group-send(REPAIR-TAG-DATA)
Await responses from majority
Select (t_{rep}, v_{rep}), for max tag
$(t_{loc}, v_{loc}) \leftarrow (t_{rep}, v_{rep})$
$status \leftarrow active$

init-repair-resp, **recv** REPAIR-TAG-DATA **from** s':
if $status = active$ **then**
Send (t_{loc}, v_{loc}) to s'

A repair operation is initiated via the action *init-repair*, by an external trigger, at a server which is in the crashed state. Note that we do not explicitly define a *crashed* state since a crash is not a part of the algorithm. We assume that as soon as the repair operation starts, the variable *status* is set to the *repair* state, and also the local (tag, value) pair is set to the default sate (t_0, v_0). The repair operation is essentially the first phase of the read operation, during which the server queries all the servers for the respective local tag and value pairs, and stores the tag and value pair corresponding to the highest tag after receiving responses from a majority of servers. Finally, the repair operation is terminated setting variable *status* to *active* state. A server in S responds to a request generated from *init-repair* phase only if it is in the active state.

4.1 Analysis of $RADON_R$

Liveness of read, write and repair operations in $RADON_R$ follows immediately if we assume condition $N1$ with $\alpha > \frac{3}{4}$. This is because liveness of any operation depends on sufficient number of responses from the servers during the various phases of the operation. From Fig. 1, we know that the maximum number of responses that is expected in any phase is $\lceil \frac{3n+1}{4} \rceil$, which is guaranteed under $N1$ with $\alpha > \frac{3}{4}$.

The tricky part is to prove atomicity of reads and writes. The proof is based on Lemma 13.16 of [23], a restatement of which can be found in [20]. Consider two completed write operations π_1 and π_2, such that, π_2 starts after the completion of π_1. For any completed write operation π, we define $tag(\pi) = t_w$, where t_w is the tag which the writer uses in the *put-data* phase. In this case, one of the requirements the algorithm needs to satisfy to ensure atomicity is $tag(\pi_2) > tag(\pi_1)$. While this fact is straightforward to prove for an algorithm like ABD, which does not have background repair, in $RADON_R$, we need to consider the effect of those repair operations that overlap with π_1, and also those that occur in between π_1

and π_2. The point to note is that such repair operations can potentially restore the contents of the repaired node such that the restored tag is less than $tag(\pi_1)$. We then need to show the absence of propagation of older tags (older than $tag(\pi_1)$) into a majority of nodes, due to a sequence of repairs which happen before π_2 decides its tag. We do this via the following two observations: 1) In Lemma 3, we show that any successful repair operation, which begins after a point of time T, always restores value to one, which corresponds to a tag which is at least as high as the minimum of the tags stored in any majority of active servers at time T. This fact is in turn used to prove a similar property for reads and writes, as well. 2) We next show (as part of proof of Theorem 5), under the assumption of $N1$ with $\alpha > 3/4$, the existence of a point of time T before the completion of π_1 such that a majority of nodes are active at T, and all of whose tags are at least as high as $tag(\pi_1)$. The two steps are together used to prove that $tag(\pi_2) > tag(\pi_1)$. A similar sequence of steps are used to show atomicity properties of read operations, as well.

For a completed read operation π, $tag(\pi) = t_r$, where t_r is the tag corresponding to the value v_r returned by the reader. For a completed repair π, $tag(\pi) = t_{rep}$, where t_{rep} is the tag corresponding to the value restored during the repair operation.

▶ **Lemma 3.** *Let β denote a well-formed execution of $RADON_R$. Suppose T denotes a point of time in β such that there exists a majority of servers $\mathcal{S}_m$, $\mathcal{S}_m \subset \mathcal{S}$ all of which are in the active state at time T. Also, let t_s denote the value of the local tag at server $s \in \mathcal{S}_m$, at time T. Then, if π denotes any completed repair or read operation that is initiated after time T, we have $tag(\pi) \geq \min_{s\in\mathcal{S}_m} t_s$. Also, if π denotes any completed write operation that is initiated after time T, we have $tag(\pi) > \min_{s\in\mathcal{S}_m} t_s$.*

▶ **Theorem 4** (Liveness)**.** *Let γ denote a well-formed execution of $RADON_R$, under the condition $N1$ with $\alpha > \frac{3}{4}$. Then every operation initiated by a non-faulty client completes.*

▶ **Theorem 5** (Atomcity)**.** *Every execution of the $RADON_R$ algorithm operating under the $N1$ network stability condition with $\alpha > \frac{3}{4}$, is atomic.*

We note that, though Lemma 3 gives a result about completed operations, condition $N1$ is not a prerequisite for the result in Lemma 3. In other words, the result in Lemma 3 holds for any completed operation, even if condition $N1$ is violated. As we will see, this is an important fact that we will use to establish atomicity of $RADON_R^{(S)}$ for any execution.

5 Algorithm $RADON_R$

In this section, we present the erasure-code based $RADON_C$ algorithm for implementing atomic memory service, and performing repair of crashed nodes. The algorithm uses $[n, k]$ MDS codes for storage. Liveness and atomicity are guaranteed under the following assumptions: 1) the $N1$ network stability condition with $\alpha \geq \frac{3n+k}{4n}$, 2) the number of write operations concurrent with a read or repair operation is at most δ. The precise definition of concurrency depends on the algorithm itself, and appears later in this section. The $RADON_C$ algorithm has significantly reduced storage and communication cost requirements than $RADON_R$, when δ is limited.

The algorithm (see Fig. 2) is a natural generalization of the $RADON_R$ algorithm accounting for the fact that we use MDS codes. The write operation has two phases, where the first phase finds the maximum tag in the system based on majority responses. During the second phase, the writer computes the coded elements for each of the n servers and uses the group-send operation to disperse them. The *group-send* operation here uses a vector of

Fig. 2 The protocols for write, reader, and any server $s_i \in \mathcal{S}$ in $RADON_C$.

write(v):

get-tag:
group-send(QUERY-TAG)
Await responses from majority
Select the max tag t^*

put-data:
$t_w = (t^*.z + 1, w)$.
code-elems $= [(t_w, c_1), \ldots, (t_w, c_n)]$, $c_i = \Phi_i(v)$
group-send(CODE-ELEMENTS, *code-elems*)
Terminate after $\lceil \frac{3n+k}{4} \rceil$ acks

read:

get-data:
group-send(QUERY-LIST)
Wait for $\lceil \frac{n+k}{2} \rceil$ *Lists*
Select the max tag, t_r, whose corresponding value, v_r, is decodable using the *Lists*.

put-data:
code-elems $= [(t_r, c_1), \ldots, (t_r, c_n)]$, $c_i = \Phi_i(v_r)$
group-send(CODE-ELEMENTS, *code-elems*)
Wait for $\lceil \frac{3n+k}{4} \rceil$ acks
Return v_r

Server $s_i \in \mathcal{S}$:

State Variables:
status $\in \{active, repair\}$, initially *active*
List $\subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$

get-tag-resp,**recv** QUERY-TAG **from writer** w:
if *status* $=$ *active* **then**
$t^* = \max_{(t,c) \in List} t$
Send t^* to w

get-data-resp, **recv** QUERY-LIST **from reader** r:
if *status* $=$ *active* **then**
Send *List* to r

put-data-resp, **recv** CODE-ELEMENTS, (t, c_i) **from** p :
if *status* $=$ *active* **then**
List $\leftarrow$ *List* $\cup \{(t, c_i)\}$
if $|List| > \delta + 1$ **then**
Retain the (tag, coded-element) pairs for the $\delta + 1$ highest tags in *List*, and delete the rest.
Send ack to p.

init-repair :
status $\leftarrow$ *repair*
group-send(REPAIR-LIST)
Wait for $\lceil \frac{n+k}{2} \rceil$ *Lists*
Find (tag, value) pairs decodable from *Lists*.
Restore local *List* via re-encoding and retaining the (tag, coded-element) pairs corresponding to at most $\delta + 1$ highest tags, from the above pairs
status $\leftarrow$ *active*

init-repair-resp, **recv** REPAIR-LIST **from server** s':
if *status* $=$ *active* **then**
Send *List* to s'

length n, where the i^{th} element denotes the message for the i^{th} server, $1 \leq i \leq n$. Each server keeps a *List* of up to $(\delta + 1)$ (tag, coded-element) pairs. Every time a (tag, coded-element) message arrives from a writer, the pair gets added to the *List*, which is then pruned to at most $(\delta + 1)$ pairs, corresponding to the highest tags. The writer terminates after getting acks from $\lceil \frac{3n+k}{4} \rceil$ servers.

During a read operation, the reader queries all servers for their entire local *Lists*, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. Once the reader receives *Lists* from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag t_r whose corresponding value v_r can be decoded using the using the coded elements in the lists. The read operation completes following a write-back of (t_r, v_r) using the *put-data* phase.

The repair operation is very similar to the first phase of the read operation, during which a server collects lists from $\lceil \frac{n+k}{2} \rceil$ servers. But this time, the server figures out the set of all the possible tags that can be decoded from among the *Lists*, and prunes the set to the highest $(\delta + 1)$ tags. The repaired *List* then consists of (tag, coded-element) pairs corresponding these (at most) $(\delta + 1)$ tags. Assuming repair of server i, the creation of a coded-element corresponding to a value v involves first decoding the value v, and then computing $\Phi_i(v)$ (referred to as re-encoding in Fig. 2).

5.1 Analysis of $RADON_C$

Throughout this section, we assume network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Tags for completed read and write operations are defined in the same manner as we did for $RADON_R$; we avoid repeating them here. We first discuss liveness properties of $RADON_C$. Let us

first consider liveness of repair operations. Towards this, note from the algorithm in Fig. 2 that a repair operation never gets stuck even if it does not find any set of k *Lists* among the responses, all of which have a common tag. In such a case, the algorithm allows the possibility that the repaired *List* is simply empty, at the point of execution when the server re-enters the active state. In other words, liveness of a repair operation is trivially proved, i.e., a server in a repair state always eventually reenters the active state, as long as it does not experience a crash during the repair operation. The triviality of liveness of repair operations, observed above, does not extend to read operations. For a read operation to complete the *get-data* phase, it must be able to find a set of k *Lists* among the responses all of which contain coded-elements corresponding to a common tag; otherwise a read operation gets stuck. The discussion above motivates the following definitions of valid read and valid repair operations.

▶ **Definition 6** (Valid Read and Repair Operations)**.** A read operation will be called as a valid read if the associated reader remains alive at least until the reception of the $\lceil \frac{n+k}{2} \rceil$ responses during the *get-data* phase. Similarly, a repair operation will be called a valid repair if the associated server does not experience a further crash event during the repair operation.

▶ **Definition 7** (Writes Concurrent with a Valid Read (Repair))**.** Consider a valid read (repair) operation π. Let T_1 denote the point of initiation of π. For a valid read, let T_2 denote the earliest point of time during the execution when the associated reader receives all the $\lceil \frac{n+k}{2} \rceil$ responses. For a valid repair, let T_2 denote the point of time during the execution when the repair completes, and takes the associated server back to the active state. Consider the set $\Sigma = \{\sigma : \sigma$ is any write operation that completes before π is initiated$\}$, and let $\sigma^* = \arg\max_{\sigma \in \Sigma} tag(\sigma)$. Next, consider the set $\Lambda = \{\lambda : \lambda$ is any write operation that starts before T_2 such that $tag(\lambda) > tag(\sigma^*)\}$. We define the number of writes concurrent with the valid read (repair) operation π to be the cardinality of the set Λ.

The above definition captures all the write operations that overlap with the read, until the time the reader has all data needed to attempt decoding a value. However, we ignore those write operations that might have started in the past, and never completed yet, if their tags are less than that of any write that completed before the read started. This allows us to ignore write operations due to failed writers, while counting concurrency, as long as the failed writes are followed by a successful write that completed before the read started.

The following lemma could be considered as the analogue of Lemma 3 for $RADON_C$. The first part of the lemma shows that under $N1$ with $\alpha \geq \frac{3n+k}{4n}$, the repaired *List* is never empty; there is always at least one (tag, coded-element) pair in the repaired *List*. Parts 2 and 3 are used to prove liveness and atomicity of client operations.

▶ **Lemma 8.** *Consider any well-formed execution β of $RADON_C$ operating under the network stability condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$. Further assume that the number of writes concurrent with any valid read or repair operation is at most δ. For any operation π, consider the set $\Sigma = \{\sigma : \sigma$ is a read or a write in β that completes before π begins$\}$, and also let $\sigma^* = \arg\max_{\sigma \in \Sigma} tag(\sigma)$. Then, the following statements hold:*

- *If π denotes a completed repair operation on a server $s \in \mathcal{S}$, then the repaired List of server s due to π contains the pair $(tag(\sigma^*), c_s^*)$.*
- *If π denotes a read operation associated with a non-faulty reader r, and further, if $\mathcal{S}_1$ denotes the set of $\lceil \frac{n+k}{2} \rceil$ servers whose responses, say $\{L_\pi(s), s \in \mathcal{S}_1\}$, are used by r to attempt decoding of a value in the get-data phase, then there exists $\mathcal{S}_2 \subseteq \mathcal{S}_1$, $|\mathcal{S}_2| = k$, such that $\forall s \in \mathcal{S}_2, (tag(\sigma^*), c_s^*) \in L_\pi(s)$.*

- *If π denotes a write operation associated with a non-faulty writer w, and further if $\mathcal{S}_1$ denotes the set of majority servers whose responses are used by w to compute max-tag in the get-tag phase, then there exists a server $s \in \mathcal{S}_1$, whose response tag $t_s \geq tag(\sigma^*)$.*

Here, c_s^* denotes the coded-element of server s for value v^*, associated with $tag(\sigma^*)$.

▶ **Theorem 9** (Liveness). *Let β denote a well-formed execution of $RADON_C$, operating under the $N1$ network stability condition with $\alpha \geq \frac{3n+k}{4n}$ and δ be the maximum number of write operations concurrent with any valid read or repair operation. Then every operation initiated by a non-faulty client completes.*

▶ **Theorem 10** (Atomicity). *Any execution of $RADON_C$, operating under condition $N1$ with $\alpha \geq \frac{3n+k}{4n}$, is atomic, if the maximum number of write operations concurrent with a valid read or repair operation is δ.*

6 The $RADON_R$ Algorithm

In this section, we present the $RADON_R^{(S)}$ algorithm having the property that every execution is atomic. Liveness is guaranteed under the slightly stronger network stability condition $N2$ with $\alpha > \frac{3}{4}$. In comparison wtih $RADON_R$, the algorithm has extra phases for both read and write operations, in order to guarantee safety of every execution.

The write operation has three phases (see Fig. 3). The first two phases are identical to those of $RADON_R$ during which the writer queries for the local tags, and then sends out the new (tag, value) pair, respectively. In the third phase, called *confirm-data*, the writer ensures the presence of at least a majority of servers, which the writer knows for sure that received its data during the second phase, *put-data*. In order to facilitate the *confirm-data* phase, the servers maintain a *Seen* variable. Any time the server receives a value from a writer, the server adds the corresponding (tag, writer ID) pair to the *Seen* list. Next, during the *confirm-data-resp* phase, the server responds to the writer only if this (tag, writer ID) pair is part of the *Seen* variable. The idea is that if the server experiences a crash and a successful repair operation in between the *put-data* and *confirm-data* phases, the server no longer has the (tag, writer ID) pair in its *Seen* variable, and hence does not respond to the *confirm-data* phase. This is because, a crash removes all state variables, including *Seen*, and the repair algorithm (see Fig. 3) simply restores the *Seen* variable to its default value, the empty set. Further, by ensuring that the writer expects acks from among a majority of servers in *confirm-data*, from among the $\frac{3n+1}{4}$ servers whose acks were obtained during *put-data*, we can guarantee that any execution is atomic.

The read operation also has three phases, first two of which are identical to those of $RADON_R$, except for the use of the *Seen* variable in the server during the *put-data* phase. The third phase is the *confirm-data* phase as in the write operation. The repair operation has one phase, and is nearly exactly identical to that of $RADON_R$. Note that the *Seen* variable gets reset to its initial value during repair.

6.1 Analysis of $RADON_R^{(S)}$

We overview the proofs of liveness and atomicity before formal claims. For liveness of writes, we assume $N2$ with $\alpha > \frac{3}{4}$, and argue the existence of a majority $\mathcal{S}_m$ of servers all of which remain active from the point of time at which the *group-send* operation gets initiated in the *put-data* phase, till the point of time all the servers in $\mathcal{S}_m$ effectively consume requests for *confirm-data* from the writer. In this case, write operation completes after receiving acks from servers in $\mathcal{S}_m$ during the *confirm-data* phase. The set $\mathcal{S}_m$ exists because, under $N2$

Fig. 3 The protocols for writer, reader, and any server $s \in \mathcal{S}$ in $RADON_R^{(S)}$.

write(v):
get-tag:
  group-send(QUERY-TAG)
  Await responses from majority
  Select the max tag t^*
put-data:
  $t_w = (t^*.z + 1, w)$.
  group-send((PUT-DATA, (t_w, v)))
  Wait for $\lceil \frac{3n+1}{4} \rceil$ acks (say from $\mathcal{S}_\alpha$)
confirm-data:
  group-send((CONFIRM-DATA, t_w))
  Terminate after acks from majority from among servers in $\mathcal{S}_\alpha$

read:
get-data:
  group-send(QUERY-TAG-DATA)
  Await responses from majority
  Select (t_r, v_r), with max tag.
put-data :
  group-send((PUT-DATA, (t_r, v_r)))
  Wait for $\lceil \frac{3n+1}{4} \rceil$ acks (say from $\mathcal{S}_\alpha$)
confirm-data:
  group-send((CONFIRM-DATA, t_r))
  Await acks from a majority of servers in $\mathcal{S}_\alpha$
  Return v_r

Server $s \in \mathcal{S}$:
State Variables:
$(t_{loc}, v_{loc}) \in \mathcal{T} \times \mathcal{V}$, initially (t_0, v_0)
$status \in \{active, repair\}$, initially *active*
$Seen \subseteq \mathcal{T} \times \{\mathcal{W} \cup \mathcal{R}\}$, initially empty

get-tag-resp, **recv** QUERY-TAG **from writer** w:
  if $status = active$ **then**
    Send t_{loc} to w
get-data-resp, **recv** QUERY-TAG-DATA **from reader** r:
  if $status = active$ **then**
    Send (t_{loc}, v_{loc}) to r
put-data-resp, **recv** (PUT-DATA, (t, v)) **from** c :
  if $status = active$ **then**
    if $t > t_{loc}$ **then**
      $(t_{loc}, v_{loc}) \leftarrow (t, v)$
    $Seen \leftarrow Seen \cup \{(t, c)\}$
    Send ack to c.

confirm-data-resp, **recv** (CONFIRM-DATA, t) **from** c:
  if $status = active$ **then**
    if $(t, c) \in Seen$ **then**
      Remove (t, c) from $Seen$
      Send ack to client c.

init-repair :
  $status \leftarrow repair$
  $(t_{loc}, v_{loc}) \leftarrow (t_0, v_0)$
  $Seen \leftarrow \emptyset$
  group-send(REPAIR-TAG-DATA)
  Await responses from majority.
  Select (t_{rep}, v_{rep}), with max tag
  $(t_{loc}, v_{loc}) \leftarrow (t_{rep}, v_{rep})$
  $status \leftarrow active$

init-repair-resp, **recv** REPAIR-TAG-DATA **from** s':
  if $status = active$ **then**
    Send (t_{loc}, v_{loc}) to s'

with $\alpha > \frac{3}{4}$, a set $\mathcal{S}_\alpha$ of $\lceil \frac{3n+1}{4} \rceil$ servers remain alive from the start of the group-send, till the effective consumption of the acks by the writer in *put-data* phase. Also, a second set $\mathcal{S}'_\alpha$ of $\lceil \frac{3n+1}{4} \rceil$ servers remain active from the start of the group-send in the *confirm-data* phase, till all servers in $\mathcal{S}'_\alpha$ complete the respective effective consumption from this group-send. We note that $\mathcal{S}'_\alpha \cap \mathcal{S}_\alpha$ is at least a majority. We next use the observation that the *group-send* operation in the *confirm-data* phase forms part of the effective consumption of the last of the acks in the *put-data* phase. Using this, we argue that the servers in $\mathcal{S}'_\alpha \cap \mathcal{S}_\alpha$ remain active till they effectively consume message from *group-send* operation of the *confirm-data* phase, and thus $\mathcal{S}'_\alpha \cap \mathcal{S}_\alpha$ is a candidate for $\mathcal{S}_m$. The liveness of read is similar to that of write, while liveness of repair is straightforward under $N2$ with $\alpha > \frac{3}{4}$.

Towards proving atomicity of reads and writes, we first define tags for completed reads, writes and repair operations exactly in the same manner as we did in $RADON_R$. Consider two completed write operations π_1 and π_2 such that π_2 starts after the completion of π_1, and we need to show that $tag(\pi_2) > tag(\pi_1)$. As in $RADON_R$, we do this in two parts: Lemma 3 holds as it is for $RADON_R^{(S)}$ as well. Recall that Lemma 3 essentially shows that if a majority of active nodes is locked-on to any particular tag, say t', at a specific point of time T during the execution of the algorithm, then any repair operation which begins after the time T always restores the tag to one which is at least as high as t'. The challenge now is to show the existence of these favorable points of time instants T as needed in the assumption

of the lemma. While in $RADON_R$, we used the $N1$ to argue this, in $RADON_R^{(S)}$, we do not use $N2$; instead we rely on the third *confirm-data* phase of the first write operation π_1.

▶ **Theorem 11** (Liveness). *Let β denote a well-formed execution of $RADON_R^{(S)}$ under condition $N2$ with $\alpha > \frac{3}{4}$. Then every operation initiated by a non-faulty client completes.*

▶ **Theorem 12** (Atomcity). *Every execution of the $RADON_R^{(S)}$ algorithm is atomic.*

7 Storage and Communication Costs of Algorithms

We give a justification of storage and communication cost numbers of the three algorithms, appearing in Table 1. Recall that the size of value v is assumed to be 1 and also that we ignore the costs due to metadata. It is clear that both $RADON_R$ and $RADON_R^{(S)}$ have storage cost n, write cost n, and read cost $2n$ (due to write back). For $RADON_C$, each server stores at most $\delta + 1$ coded-elements, where each element has size $\frac{1}{k}$. Thus storage cost of $RADON_C$ is $(\delta + 1)\frac{n}{k}$. The write cost of $RADON_C$ is simply $\frac{n}{k}$, and the contribution comes from the writer sending one coded-element to each of the n servers. For a read, getting the entire *Lists* during the $get - data$ phase incurs a cost of $(\delta + 1)\frac{n}{k}$. The write-back phase incurs an additional cost of $\frac{n}{k}$. Thus, the total read cost in $RADON_C$ is $(\delta + 2)\frac{n}{k}$.

8 Conclusions

In this paper, we provided an erasure-code-based algorithm for implementing atomic memory, having the ability to perform repair of crashed nodes in the background, without affecting client operations. We assumed a static model with a fixed, finite set of nodes, and also a practical network condition $N1$ to facilitate repair. We showed how the usage of MDS codes significantly improve storage and communication costs over a replication based solution, when the number of writes concurrent with a read or repair is limited. Liveness and atomicity are guaranteed as long as $N1$ is satisfied; however violation of $N1$ can lead to non-atomic executions. We further showed how a slightly stringent network condition $N2$ can be used to construct a replication based algorithm that always guarantees atomicity. Ongoing efforts include exploring possibility of using repair-efficient erasure codes [11] in $RADON_C$, and testbed evaluations on cloud based infrastructure.

References

1 M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 336–345, 2005.

2 M. K. Aguilera, I. Keidar, D. Malkhi, J. P. Martin, and A. Shraery. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–081, 2010.

3 M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, pages 7:1–7:32, 2011.

4 H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.

5 H. Attiya, H. C. Chung, F. Ellen, S. Kumar, and J. L. Welch. Simulating a shared register in an asynchronous system that never stops changing - (extended abstract). In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 75–91, 2015.

6 R. Baldoni, S. Bonomi, A. M. Kermarrec, and M. Raynal. Implementing a register in a dynamic distributed system. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 639–647, June 2009.

7 C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 115–124, 2006.

8 V. R. Cadambe, N. A. Lynch, M. Médard, and P. M. Musial. A coded shared atomic memory algorithm for message passing architectures. In *Proceedings of 13th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 253–260, 2014.

9 F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, jun 2008.

10 G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 205–220, New York, NY, USA, 2007. ACM.

11 A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

12 D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić. Powerstore: proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 285–298, 2013.

13 P. Dutta, R. Guerraoui, and R. R. Levy. Optimistic erasure-coded distributed storage. In *Proceedings of the 22nd international symposium on Distributed Computing (DISC)*, pages 182–196, Berlin, Heidelberg, 2008.

14 R. Fan and N. Lynch. Efficient replication of large data objects. In *Distributed algorithms*, Lecture Notes in Computer Science, pages 75–91, 2003.

15 R. Guerraoui, R. R. Levy, B. Pochon, and J. Pugh. The collective memory of amnesic processes. *ACM Trans. Algorithms*, 4(1):1–31, 2008.

16 J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.

17 C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 15–26, 2012.

18 W. C. Huffman and V. Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2003.

19 K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Medard, and A. A. Schwarzmann. Storage-optimized data-atomic algorithms for handling erasures 124 and errors in distributed storage systems. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2016.

20 K. M. Konwar, N. Prakash, M. Medard, and N. Lynch. RADON: Repairable atomic data object in networks. *CoRR*, abs/1605.05717, 2016.

21 L. Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

22 N. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)*, pages 173–190, 2002.

23 N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

24 Peter Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Communications of the ACM*, 57(6):88–98, 2014.

25 K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST)*, pages 81–94, 2015.

26 I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

27 M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*, pages 325–336, 2013.

28 C. Shao, J. L. Welch, E. Pierce, and H. Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011.

29 A. Spiegelman, Y. Cassuto, G. Chockler, and I. Keidar. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS2015)*, 2015.

30 A. Spiegelman and I. Keidar. On liveness of dynamic storage. *CoRR*, abs/1507.07086, 2015. URL: `http://arxiv.org/abs/1507.07086`.

31 A. Spiegelman, I. Keidar, and D. Malkhi. Dynamic reconfiguration: A tutorial. *OPODIS 2015*, 2015.

Computationally Light "Multi-Speed" Atomic Memory

Antonio Fernández Anta[1], Theophanis Hadjistasi[2], and Nicolas Nicolaou[3]

1 **IMDEA Networks Institute, Madrid, Spain**
antonio.fernandez@imdea.org

2 **University of Connecticut, Storrs, CT, USA**
theophanis.hadjistasi@uconn.edu

3 **IMDEA Networks Institute, Madrid, Spain**
nicolas.nicolaou@imdea.org

Abstract

Communication demands are usually the leading factor that defines the efficiency of operations on a read/write shared memory emulation in the message-passing environment. In the quest for minimizing the communication demands, the algorithms proposed either require restrictions in the system or incur high computation demands. As a result, such solutions may be not suitable to be used in practice.

In this paper we focus on the practicality of implementations of *atomic read/write shared memory* emulation in the message-passing environment. In particular we investigate implementations that reduce both *communication* and *computation* demands. We first examine the shortcomings of the best two (in terms of communication demands) known algorithms that implement atomic single-writer multiple-reader (SWMR) atomic memory, [3, 6]. The algorithm ccFast proposed in [3], achieves optimal communication by allowing each operation to complete in one round trip, with light computation requirements. Unfortunately, it relies on strict limitations on the number of readers. On the other hand, algorithm OhSam [6], imposes no restrictions on the system, but provides operations that require one and a half communication rounds. In the light of these shortcomings, we present two algorithms that implement *multi-speed* operations with *light computation,* and *without imposing* any restriction on the system. In particular, algorithm ccHybrid adopts the fast (one-round) writes presented in [3], and makes *clients* to switch to a slow (two-round) mode whenever the system is congested. On the other hand, algorithm OhFast, pushes the responsibility of deciding for the speed switch to the *servers*. This allows the algorithm to utilize the fast operations presented in [3], and the slow one-and-a-half-rounds operations of [6], whenever is necessary. We prove that both new algorithms preserve atomicity. To evaluate the new algorithms we implement five different atomic memory algorithms in the NS3 simulator, and we compare their performance in terms of *operation latency*, and *ratio of slow over fast operations* performed. We test the algorithms over different: (i) topologies, and (ii) operation loads. Our results support that the newly presented algorithms increase the practicality of atomic read/write atomic shared memory implementations in the message-passing, asynchronous environment.

1998 ACM Subject Classification C.3.4 Distributed Systems, C.4 Performance of Systems

Keywords and phrases atomicity, read/write objects, shared memory, operation latency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.29

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 29; pp. 29:1–29:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Emulating atomic [8] (linearizable [7]) read/write objects in message-passing environments is one of the fundamental problems in distributed computing. The problem becomes more difficult when participants in the service may fail and the environment is asynchronous, i.e., it cannot provide any time guarantees on the delivery of the messages and the computation speeds. To cope with failures, traditional distributed object implementations like [1, 10], use *redundancy* by replicating the object to multiple (possibly geographically dispersed) locations (replica servers). Replication however raises the challenge of consistency, as multiple object copies can be accessed concurrently by multiple processes. Atomicity is the most intuitive consistency semantic, as it provides the illusion of a single-copy object that serializes all accesses: each read operation returns the value of the latest write operation.

Attiya, Bar-Noy, and Dolev [1] were the first to present an algorithm, known as ABD, to implement single-writer multi-reader (SWMR) atomic objects in message-passing, crash-prone, asynchronous environments. The authors associate logical *timestamps* to the values written, to impose an order on the write operations. The propagation of the latest timestamp (and its corresponding value) is based on the assumption that at least a majority of replica servers do not fail. In this setting, ABD has write operations that terminate with a single communication round-trip, and read operations that involve two round-trips. Based on basic value comparisons, ABD incurs almost no computational overhead to the service participants. Atomicity is guaranteed by the intersecting properties of two majorities and the second phase of a read operation. Following ABD, a folklore belief persisted that in asynchronous multi-reader (MR) atomic memory algorithms, "reads must write."

The work by Dutta et al. [2] refuted this belief, by presenting atomic register algorithms in which every operation involves only a *single* round-trip. Such an algorithm is called *fast*. They showed that fast reads are possible only in the single-writer (SW) model, and given that the number of readers R is constrained with respect to the number of replicas S and the maximum number of failures f; in particular, $R < \frac{S}{f} - 2$. A recent work by Fernández Anta, Nicolaou, and Popa [3], has shown that, although the result in [2] is efficient in terms of communication, it requires processes to evaluate a computationally hard (NP-hard) predicate. A new algorithm ccFast, with a new predicate, was proposed in that paper to allow operations terminate with a linear computation overhead. Despite improving the practicality of [2], the algorithm in [3] inherited the same system constraint as [2].

The idea of exploring "multi-speed" read operations is not new. An algorithm is said to be "multi-speed" when different read operations may perform different number of communication rounds before completing. Works like [4, 5] proposed implementations in the SWMR model with *two-speed* operations, in an attempt to relax the constraints proposed in [2], and to allow unbounded number of readers. In particular, the work in [5] presents algorithm Sf, which applies a predicate similar to the one introduced in [2], but on *virtual nodes* (i.e., sets of readers) instead of individual reader processes. In [4], the authors introduced *quorum views*, which are client-side tools that examine the distribution of the latest value among the replicas, in order to enable fast read operations. Both [4, 5] trade communication for scalability. Under conditions of low concurrency, both algorithms allow most reads to complete in a single communication round-trip; otherwise a *two round-trip* operation (similar to ABD) is required. To determine the speed of an operation, both algorithms inflicted significant computational demands: (i) [5] exploited the same predicate as in [2], which is NP-hard [3], and (ii) [4] needed to examine the distribution of the object value within all the possible replica subsets. Thus, a trend appeared in the algorithms that aimed for fast operations:

algorithms with lower communication rounds demanded higher computation overhead at the processes.

Following the above findings, we say that an operation is *fast* if it completes in a single communication round trip, and *slow* if it completes in two round trips. A recent work by Hadjistasi, Nicolaou and Schwarzmann [6] redefines *slowness*, as they present an algorithm for the SWMR model, called OhSam, where each operation takes *one and a half* round-trips to complete. As the number of readers is bounded when all operations are fast [2], the authors claim the optimality of their approach in terms of communication when no constraint is imposed. Furthermore their algorithm relies on basic comparisons, inflicting negligible computation overhead.

Contributions. In this paper, we focus in improving the practicality of SWMR atomic read/write register algorithms, by achieving low communication and computation costs on the atomic operations. We trade communication for scalability, by adopting the predicate presented in [3] and allowing some operations to be slow. Also, we combine ideas presented in both [3] and [6], to introduce implementations that allow only *single* and *one-and-a-half* round operations. Enumerated, our contributions are the following:

- We introduce a new "multi-speed" algorithm, ccHybrid, that allows operations to terminate in *one* or *two* communication round-trips, and does not impose any bounds on the number of readers. ccHybrid uses the predicate introduced in [3] to determine the speed of a read operation, and it requires at most one *complete* slow operation per written value. This is similar to the semifast algorithm Sf [5]. However, in contrast to Sf, in which processes have to decide NP-hard predicates, it incurs light (linear) computation.
- Next we examine whether we can combine the techniques presented in [3] and [6] to obtain a "multi-speed" algorithm that allows *one* and *one-and-a-half* round-trip operations. We present algorithm OhFast, that achieves the targeted performance by moving the decision on whether a slow read operation is necessary to the servers. When servers determine that a slow read is necessary, they perform a *relay* phase to inform other servers before replying to the reader. It is interesting that in OhFast not all the servers need to perform a relay for a single read operation. Some of the servers may reply directly to the read whereas some others may perform a relay phase for the same read. Thus a read operation may terminate before receiving a reply from a relaying server.
- We complement our algorithms with experimental results for five algorithms: ABD, OhSam, ccHybrid, OhFast, and Sf. ABD sets the threshold for the rest of the algorithms, while OhSam sets the threshold on the operations that use one and a half rounds. Algorithm Sf is used to demonstrate whether computation has an impact to the latency of operations. We test our algorithms under different scenarios by changing the number of participants, the frequency of operations, and using two network topologies: (i) a topology where servers are distributed evenly over the network, and (ii) a topology that resembles a datacenter where servers are concentrated in close proximity and communicate through high bandwidth links. Our results show that the proposed algorithms outperform the algorithms with "one speed" operations (i.e., ABD and OhSam) in all scenarios, reducing the latency per operation to less than half in most cases. Compared with the semifast "multi-speed" algorithm Sf, our algorithms achieve a similar read latency, even though the scenarios explored were extremely favorable for Sf, since we observed that practically all its operations were fast and the NP-hard predicate evaluations were not heavy (mainly due to the good communication conditions). Finally, as expected, we observed that the topology has a great impact on the algorithms that use *one and a half* round operations.

2 Model

We assume a system consisting of three distinct sets of processes: a writer process with identifier w, a set $\mathcal{R}$ of readers, and a set $\mathcal{S}$ of replica servers. Let $\mathcal{I} = \{w\} \cup \mathcal{R} \cup \mathcal{S}$. In a read/write object implementation, we assume that the object may take a value from a set V. The writer is the sole process that is allowed to modify the value of the object, the readers are allowed to obtain the value of the object, and each server maintains a copy of the object to ensure the availability of the object in case of failures. We assume an *asynchronous* environment, where processes communicate by exchanging messages. The writer, any subset of readers, and up to $f < \frac{|\mathcal{S}|}{2}$ servers may *crash* without any notice.

An algorithm A is a collection of processes, where process A_p is assigned to processor $p \in \mathcal{I}$. Each processor p has a *state* which is determined over a set of state variables. The state of A is a vector that contains the state of each process. Algorithm A performs a *step*, when some process p atomically:

(i) receives a message,
(ii) performs local computation,
(iii) sends a message.

Each such step causes the state at p to change from a pre-state σ_p to a post-state σ'_p. Hence, the state of A changes from σ to σ' where σ contains state σ_p for p and σ' contains state σ'_p, while the state of every $p' \neq p$ is the same in both σ and σ'. An *execution fragment* is an alternating sequence of states and actions of A ending in a state. An *execution* is an execution fragment that starts with the initial state. An execution fragment ξ' extends an execution fragment ξ if the last state of ξ is the first state of ξ'. A process p *crashes* in an execution if it stops taking steps; otherwise p is *correct*. Each process may perform a read or write operation, and each operation has *invocation* and *response* steps. An operation π is *complete* in an execution ξ, if ξ contains both the invocation and the *matching* response step for π; otherwise π is *incomplete*. An execution ξ is *well formed* if any process p that invokes an operation π in ξ does not invoke any other operation π' before the matching response step of π appears in ξ. An operation π *precedes* an operation π' in an execution ξ, denoted by $\pi \to \pi'$, if the response step of π appears before the invocation step of π' in ξ. Two operations are *concurrent* if none precedes the other.

Correctness of an implementation of an atomic read/write object is defined in terms of the *atomicity* and *termination* properties. The termination property requires that any operation invoked by a correct process eventually completes. For atomicity we use the definition of [9, Lemma 13.16].

Efficiency Metrics. We measure the complexity of an operation π in terms of:

(i) *message complexity*, i.e. the worst-case number of messages exchanged during π, and
(ii) *operation latency*, i.e. the *computation time* and the *communication delays* incurred by π. Computation time accounts the computation steps the algorithm performs in each operation.

Communication delays are measured in *communication exchanges*, as defined in [6].

In particular, a protocol requires each operation to involve a sequence of sends (or broadcasts) of typed messages and the corresponding receives. A *communication exchange* during an operation π in an execution ξ, is defined as the collection of send and receive actions for a specific typed message (as required by the protocol) between the invocation and response of π in ξ. Using this definition, implementations, such as ABD, are structured in terms of *rounds*, where each round consists of two message exchanges: a *broadcast*, initiated

by the process executing an operation, and a *convergecast* of responses to the initiator. A *fast* operation as in [5, 2] consists of two communication exchanges (or one round), and a *slow* operation as used in [1, 4, 5] consists of four communication exchanges (or two rounds). A read operation as in [6] consists of three communication exchanges (or 1.5 rounds). The number of messages that a process expects during a convergecast depends on the implementation.

3 State-of-the-Art Performance of Atomic Memory Implementations

The algorithm by Dutta et al. in 2004 [2], we refer to it as FAST, was the first to present atomic register implementations where all operations take a *single* communication round before completing. To allow fast reads, FAST deploys a recording mechanism at each server and evaluates a predicate at each reader. It was shown that fast reads are possible only if $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$ readers participate in the service. To avoid the bound on the number of readers, Georgiou, Nicolaou and Shvartsman [5], grouped the readers under logical sets, they called *virtual groups*, and allowed some of the read operations to perform two rounds (or 4 communication exchanges). The predicate of [2] was applied on the virtual groups instead of individual readers, where each group could have an arbitrary size. As expected, the use of the predicate imposed a bound on the number of virtual nodes, for atomicity to be preserved; that is $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 1$.

Fernández Anta, Nicolaou and Popa [3], showed that the predicate used by both [2] and [5], is computationally hard. This was due to the fact that the original predicate was searching among all the subsets of servers to identify if there is some subset of servers that replied to a "large enough" subset of readers. To avoid this computational overhead, they investigate whether it is possible to use *how many* instead of *which* readers obtained the latest value, and still be able to preserve atomicity. Thus, the paper introduced a new algorithm, called CCFAST, that was using the following predicate at the readers:

$$\exists \alpha \in [1, |\mathcal{R}|+1] \text{ s.t. } MS = \{s : (s,m) \in maxAck \ \wedge \ m.views \geq \alpha\} \text{ and } |MS| \geq |\mathcal{S}| - \alpha f \,.$$

Essentially, each server records the readers that observed its local timestamp in a set *seen*, and whenever requested, it reports the *cardinality* of that set to the requesting process. A reader collects the replies from the servers in each read operation, detects the replies that contain the maximum timestamp (set *maxAck*), and checks the cardinalities reported in those replies (*m.views*). If there are "enough" replies with "sufficiently" large cardinalities, the predicate holds and the reader returns the value associated with the maximum timestamp; otherwise the value associated with the previous timestamp is returned. The evaluation of the predicate can be done in linear time with respect to the number of servers in the system. Their algorithm inherited the necessary bound presented in [2] on the number of readers participants, $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$.

Finally, Hadjistasi, Nicolaou and Schwarzmann [6] closed the gap of the communication of read/write operations by presenting algorithm OHSAM, where writes take just one round (or 2 communication exchanges) and reads always take one and a half round (or 3 communication exchanges) to complete. The main idea of the algorithm is to allow servers to exchange information about the operations, before replying to the invoking process. OHSAM uses negligible computation at the processors, as each operation performs only basic comparisons. However, the server communication in *every* operation makes the algorithm suitable for environments where server communication is being carried out by high capacity links.

Table 1, summarizes the efficiency of each of the algorithms in different efficiency metrics. It also presents any bounds that an algorithm may impose on the participation of the service

Table 1 Communication, Computation, Message Complexities and Participation Bounds. (WE/RE: write/read-communication excanges, WC/RC: write/read-computation, WM/RM: write/read-number of messages). $\mathcal{V}$ is the set of virtual nodes.

Algorithm	WE	RE	WC	RC	WM	RM	Bounds
ABD [1]	2	4	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$4\lvert\mathcal{S}\rvert$	Unbounded
Fast [2]	2	2	$O(1)$	$O(\lvert\mathcal{S}\rvert^2 \cdot 2^{\lvert\mathcal{S}\rvert})$	$2\lvert\mathcal{S}\rvert$	$2\lvert\mathcal{S}\rvert$	$\lvert\mathcal{R}\rvert < \frac{\lvert\mathcal{S}\rvert}{f} - 2$
ccFast [3]	2	2	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$2\lvert\mathcal{S}\rvert$	$\lvert\mathcal{R}\rvert < \frac{\lvert\mathcal{S}\rvert}{f} - 2$
Sf [5]	2	2 or 4	$O(1)$	$O(\lvert\mathcal{S}\rvert^2 \cdot 2^{\lvert\mathcal{S}\rvert})$	$2\lvert\mathcal{S}\rvert$	$O(4\lvert\mathcal{S}\rvert)$	$\lvert\mathcal{V}\rvert < \frac{\lvert\mathcal{S}\rvert}{f} - 1$
OhSam [6]	2	3	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$2\lvert\mathcal{S}\rvert + \lvert\mathcal{S}\rvert^2$	Unbounded
ccHybrid (here)	2	2 or 4	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$O(4\lvert\mathcal{S}\rvert)$	Unbounded
OhFast (here)	2	2 or 3	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$O(\lvert\mathcal{S}\rvert^2)$	Unbounded

Algorithm 1 Write, Read and Server protocols of algorithm ccHybrid.

```
1: at the writer w
2: Components:
3: ts ∈ ℕ+; v, vp ∈ V; wcounter ∈ ℕ+
4: Initialization:
5: ts ← 0; v ← ⊥; vp ← ⊥; wcounter ← 0
6: function WRITE(val)
7:   vp ← v; v ← val
8:   ts ← ts + 1
9:   wcounter ← wcounter + 1
10:  send(⟨ts, v, vp⟩, w, wcounter) to all servers
11:  wait until |S| − f servers reply
12:  return(OK)
13: end function

14: at each reader r_i
15: Components:
16: ts ∈ ℕ+; maxTS ∈ ℕ+; v, vp ∈ V; rcounter ∈ ℕ+
17: srvAck ⊆ S × M
18: Initialization:
19: ts ← 0; maxTS ← 0; v ← ⊥; vp ← ⊥, rcounter ← 0
20: function READ( )
21:   rcounter ← rcounter + 1
22:   send(⟨ts, v, vp⟩, r_i, rcounter) to all servers
23:   wait until |S| − f servers reply
      ▷ Collect (sid, ⟨ts′, v′, vp′⟩, views, prop)) msgs in srvAck
24:   maxTS ← max({m.ts′ | (s, m) ∈ srvAck})
25:   maxAck ← {(s, m) | (s, m) ∈ srvAck ∧ m.ts′ = maxTS}
26:   ⟨ts, v, vp⟩ ← m.⟨ts′, v′, vp′⟩ for (∗, m) ∈ maxAck
27:   maxViews ← max({m.views | (s, m) ∈ maxAck})
28:   propSet ← {s | (s, m) ∈ maxAck ∧ m.prop = True}
29:   if maxViews > |S|/f − 2 ∨ propSet ≠ ∅ then
30:     if |propSet| < f + 1 then
                                              ▷ Phase 2
31:       send(⟨ts, v, vp⟩, r_i, rcounter) to all servers
32:       wait until |S| − f servers reply
33:     end if
34:     return(v)
35:   else
36:     if ∃α ∈ [1, |S|/f − 2] s.t.
37:       MS = {s : (s, m) ∈ maxAck ∧ m.views ≥ α} and
38:       |MS| ≥ |S| − αf then
39:       return(v)
40:     else
41:       return(vp)
42:     end if
43:   end if
44: end function

45: at each server s_i
46: Components:
47: ts ∈ ℕ+; seen ⊆ R ∪ {w}; v, vp ∈ V; prop ∈ {True, False}
48: Counter[|R| + 1] ∈ ℕ+
49: Initialization:
50: ts ← 0; seen ← ∅; v, vp ← ⊥; prop ← False
51: Counter[i] ← 0 for i ∈ R ∪ {w}
52: function RCV(⟨ts′, v′, vp′⟩, q, counter)
                          ▷ Called upon reception of a message
53:   if Counter[q] < counter then
54:     if ts′ > ts then
55:       ⟨ts, v, vp⟩ ← ⟨ts′, v′, vp′⟩
56:       seen ← {q}
57:       prop ← False
58:     else
59:       seen ← seen ∪ {q}
60:     end if
61:     if ts′ = ts ∧ q ∈ R then
62:       prop ← True
63:     end if
64:     send(⟨ts, v, vp⟩, |seen|, prop) to q
65:   end if
66: end function
```

in order to be able to provide atomic guarantees. The last two algorithms are the ones we present in this paper. Notice that the goal is to minimize communication without inflicting high computation overheads, or participation bounds in the system.

4 Algorithm ccHybrid: Switching from One to Two Rounds

As discussed in Section 3, algorithm ccFast guarantees correctness only when the number of readers is bounded with respect to the ratio of the number of servers and the number of failures in the system, i.e. $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 1$. In this section we propose a modification to ccFast that removes the bound on the number of readers. To unleash the number of readers, the new algorithm ccHybrid, allows some read operations to complete in two rounds. In particular, ccHybrid combines ideas from ccFast and ABD:

(i) it exploits *timestamp-value* pairs to order the write operations,

(ii) it uses the predicate proposed by ccFast to determine the value returned by a *fast* read, and

(iii) propagates the maximum *timestamp-value* pair to a majority of servers during a *slow* read.

The biggest challenge in ccHybrid is to determine *when* a second phase is necessary, and ensure that such a strategy does not violate atomicity. The idea of ccHybrid is to have the reader examine if the number of processes that observed the latest value is over the bound $\frac{|\mathcal{S}|}{f} - 1$. If not, then ccHybrid evaluates the predicate proposed in ccFast over the replies, to determine the value to return. Otherwise, it proceeds to a propagation phase to send the latest value to a majority of servers. To prevent readers from propagating an already propagated value, servers maintain a flag that indicates whether a timestamp has been propagated.

Algorithm 1 provides the formal pseudocode of ccHybrid. The write protocol remains the same as in both ccFast and ABD: the writer increments its local timestamp (L8) and propagates the *timestamp-value* pair to a majority of servers (L10-11). The server protocol is more involved. In addition to the replica state (timestamp and value), a server s maintains a set *seen* to record the processes that requested this replica, and a flag *prop* that, as we explain later, optimizes read operations. A server s waits for read and write requests. When a request is received, s updates its local *timestamp-value* pair (L51-57) if the *timestamp* attached in the received message is greater than its local timestamp. In addition, it initializes its *seen* set to contain the sender process, and sets the *prop* flag to $False$. In case the timestamp of the message is not greater than the local timestamp of s, then the server records the sender in its *seen* set (L59). The server s sets $prop = True$ when it receives a message from a reader that contained a *timestamp-value* pair equal to the one that is locally stored in s. Notice that a reader propagates a *timestamp-value* pair in every phase. So, s may set *prop* during the first or second phase of a read.

The main departure of ccHybrid from ccFast lies in the read protocol. A reader behaves as in ccFast as long as the maximum number of *views* reported by the servers remains below $\frac{|\mathcal{S}|}{f} - 2$. In particular, a reader sends read messages to all the servers and waits from $|\mathcal{S}| - f$ to reply (L22). When those replies are received, the reader discovers the maximum timestamp ($maxTS$) among the replies (L24), the set of messages that contained $maxTS$ (L25), and the maximum views reported in those messages (L27). If the maximum views are less than $\frac{|\mathcal{S}|}{f} - 2$ and no reader propagated $maxTS$ (L29), then the reader evaluates the predicate as in ccFast to decide which value to return; otherwise the reader returns the value associated with the $maxTS$. If at least $f + 1$ of the messages that contain $maxTS$, also contain $prop = True$, the reader returns without further action. If this is not the case then the reader performs a second phase propagating the maximum *timestamp-value* pair to $|\mathcal{S}| - f$ servers (L30–33). Notice that ccHybrid performs equally to ccFast when the number of readers that return the same value (not necessarily the same readers for each value) satisfies the bound required by ccFast. In any other case, a *single complete, slow* read operation (similar to [5]) is necessary per write operation. The use of the *prop* flag allows any read that succeeds a slow read, and returns the same value, to be *fast*, as:

(i) The slow read propagates the $maxTS$ to $|\mathcal{S}| - f$ servers,
(ii) a succeeding read receives replies from $|\mathcal{S}| - f$ servers, and
(iii) the read discovers $prop = True$ for $maxTS$ in more than $|\mathcal{S}| - 2f > f + 1$ servers.

4.1 Algorithm Correctness

Our algorithm is correct if it can satisfy Termination (liveness condition) and Atomicity (safety condition). It is trivial to see that termination is satisfied given that the system respects our failure model. To proof atomicity we are going to express atomicity in terms of timestamps written and returned in a SWMR model, as also presented in [3]:

A1. For each process p the ts variable is non-negative and monotonically nondecreasing.
A2. If a read ρ succeeds a write operation $\omega(ts)$ and returns a timestamp ts', then $ts' \geq ts$.
A3. If a read ρ returns ts', then either a write $\omega(ts')$ precedes ρ, i.e. $\omega(ts') \to \rho$, or $\omega(ts')$ is concurrent with ρ.
A4. If ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$ and ρ_1 returns ts_1, then ρ_2 returns $ts_2 \geq ts_1$.

Due to space limitations and due to the similarity of the writer and server protocols to the ones used in ccFast, we omit some of the proofs and we refer the reader to specific lemmas presented in [3]. Properties **A1** and **A3** can be extracted easily from the algorithm. Now let us proof an important lemma about the timestamp returned by a server process:

▶ **Lemma 1.** *In any execution ξ of the algorithm, if a server s receives a timestamp ts at time T from a process p, then s replies with a timestamp $ts' \geq ts$ at any time $T' > T$.*

To following lemma shows **A2**, after which we show that property **A4** holds.

▶ **Lemma 2.** *In any execution ξ of the algorithm, if a read ρ from r_1 succeeds a write operation ω that writes timestamp ts from the writer w , i.e. $\omega \to \rho$, and returns a timestamp ts', then $ts' \geq ts$.*

▶ **Lemma 3.** *In any execution ξ of* ccHybrid*, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$, ρ_1 is fast satisfying the predicate for $maxTS = ts_1$, then ρ_2 receives a $maxTS = ts_2$ s.t. $ts_2 \geq ts_1$.*

▶ **Lemma 4.** *In any execution ξ of* ccHybrid*, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$, and ρ_1 returns ts_1, then ρ_2 returns $ts_2 \geq ts_1$.*

Proof. A read operation has two modes: fast and slow. Thus, we need to examine all the possible combinations of the speeds of ρ_1 and ρ_2. There are four cases to investigate:
(a) ρ_1 is *fast*, and ρ_2 is *fast*,
(b) ρ_1 is *fast*, and ρ_2 is *slow*,
(c) ρ_1 is *slow*, and ρ_2 is *slow*, and
(d) ρ_1 is *slow*, and ρ_2 is *fast*.
Let $maxTS_i$ be the maximum timestamp observed by a read ρ_i, for $i \in \{1, 2\}$, during its first phase.

Case a: In case both operations are *fast* then, according to ccHybrid, either they observe $maxViews \leq \frac{|\mathcal{S}|}{f} - 2$ and $propSet = \emptyset$, or they observe an $|propSet| \geq f + 1$. If both observe $maxViews \leq \frac{|\mathcal{S}|}{f} - 2$ and check the predicate, then with the same reasoning as in [3, Lemma 8], it follows that $ts_2 \geq ts_1$.

If ρ_1 observes $|propSet| \geq f + 1$ then since ρ_2 receives replies from $|\mathcal{S}_2| = |\mathcal{S}| - f$ servers, then there exists a server $s \in propSet \cap \mathcal{S}_2$ such that s replies to both ρ_1 and ρ_2. Since $\rho_1 \to \rho_2$, then s replies to ρ_1 before replying to ρ_2. Since s replies with $maxTS_1$ to ρ_1, then by Lemma 1, s replies with a timestamp $ts_s \geq maxTS_1$ to ρ_2. So $maxTS_2 \geq ts_s$ and hence $maxTS_2 \geq maxTS_1$. If $maxTS_2 = maxTS_1$ then s will reply with $ts_s = maxTS_1$ and $prop = True$. In this case ρ_2 will return $ts_2 = maxTS_1 = ts_1$. If $maxTS_2 > maxTS_1$ then ρ_2 returns either $maxTS_2$ or $maxTS_2 - 1$ and thus $ts_2 \geq ts_1$.

It remains to examine the case where ρ_1 observes $maxViews \leq \frac{|\mathcal{S}|}{f} - 2$ and $propSet = \emptyset$, and ρ_2 observes $|propSet| \geq f+1$. If the predicate holds for ρ_1 then by Lemma 3, ρ_2 observes $maxTS_2 \geq maxTS_1$. Since ρ_2 observes $|propSet| \geq f+1$ then it returns $ts_2 = maxTS_2$, and

thus $ts_2 \geq ts_1$. If the predicate does not hold for ρ_1 then we know that the write operation propagating $maxTS_1 - 1$ completed before or during ρ_1. Since $\rho_1 \rightarrow \rho_2$ then this write completed before ρ_2 as well. Thus, by **A2**, ρ_2 observes $maxTS_2 \geq maxTS_1 - 1$. Since ρ_2 observes $|propSet| \geq f+1$, then it returns $ts_2 = maxTS_2 \Rightarrow ts_2 \geq maxTS_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Case b: Since ρ_1 in this case is *fast* then ρ_1 returns either: (i) $maxTS_1 - 1$, or (ii) $maxTS_1$.

In (i), since ρ_1 observed $maxTS_1$ and since we have a single writer, it follows that the write operation that wrote timestamp $maxTS_1 - 1$, say ω_1, proceeds or is concurrent to ρ_1, and completes before the response step of ρ_1. Since $\rho_1 \rightarrow \rho_2$, then $\omega_1 \rightarrow \rho_2$. Since ρ_2 is slow, then it returns the maximum timestamp it observes, i.e. $ts_2 = maxTS_2$. Moreover, since $\omega_1 \rightarrow \rho_2$, and since both operations wait for $|\mathcal{S}| - f$ replies, then according to our failure model, there exist at least a single server s that replies to both operations, first to ω_1 and then to ρ_2. According to Lemma 1, s sends a timestamp $ts_s \geq maxTS_1 - 1$ to ρ_2. Thus, $maxTS_2 \geq maxTS_1 - 1$, and therefore $ts_2 \geq ts_1$.

In (ii) it follows that either the predicate holds for ρ_1, or ρ_1 observes $|propSet| \geq f+1$. Since ρ_2 is slow and returns $ts_2 = maxTS_2$, then by Lemma 3 and with similar reasoning as in Case (a) for when ρ_1 observes $|propSet| \geq f+1$, we can show that $maxTS_2 \geq maxTS_1$ and hence $ts_2 \geq ts_1$.

Case c: The case where both reads are slow is simple and resembles the behavior of the reads in ABD [1]. Here each read ρ_i, for $i \in [1,2]$, returns $maxTS_i$ and before completing it propagates $maxTS_i$ to $|\mathcal{S}| - f$ servers. Thus, ρ_1 returns $ts_1 = maxTS_1$, and before completing propagates $maxTS_1$ to $|P_1| = |\mathcal{S}| - f$ servers. Since $\rho_1 \rightarrow \rho_2$, and since ρ_2 receives $|S_2| = |\mathcal{S}| - f$ replies, then it is going to receive a timestamp $ts_s \geq maxTS_1$ from at least a single server $s \in P_1 \cap S_2$. Thus, ρ_2 returns $ts_2 = maxTS_2 \geq maxTS_1$, and $ts_2 \geq ts_1$.

Case d: So it remains to investigate the case where ρ_1 is *slow* and ρ_2 is *fast*. Observe that this case is possible when a server s is "saturated" by concurrent reads (more than $\frac{|\mathcal{S}|}{f} - 2$) and s replies to ρ_1 but does not reply to ρ_2. Now we have two cases to investigate: either ρ_2 observes $maxTS_2 \geq maxTS_1$, or $maxTS_2 = maxTS_1 - 1$. If ρ_2 observes a $maxTS_2 \geq maxTS_1$, it may either return $ts_2 = maxTS_2$ or $ts_2 = maxTS_2 - 1$. In either case $ts_2 \geq maxTS_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Let us examine now the case where $maxTS_2 = maxTS_1 - 1$. Since ρ_1 is slow and returns $maxTS_1 - 1$, then before completing it propagates $maxTS_1 - 1$ to $|\mathcal{S}| - f$ servers. Let P_1 be the set of servers that received the messages and replied to the second phase of ρ_1. Moreover, $|S_2| = |\mathcal{S}| - f$ are the servers that received messages and replied to ρ_2. So by Lemma 1, every server $s \in P_1 \cap S_2$ replies to both ρ_1 and then to ρ_2, with a timestamp $ts_s \geq maxTS_1 - 1$. In addition s sets $prop = True$ before replying to ρ_1. Since $maxTS_2 = maxTS_1 - 1$, then s replies with $ts_s = maxTS_1 - 1$ to ρ_2, and thus the $propSet$ contains at least s in ρ_2. According to the algorithm ρ_2 returns $ts_2 = maxTS_2$ in this case and hence $ts_2 \geq ts_1$. ◀

▶ **Theorem 5.** *Algorithm* ccHybrid *implements a SWMR atomic read/write register.*

5 Algorithm OhFast: Switching from One to One and a Half Rounds

Similar to algorithm ccHybrid, OhFast aims to allow unbounded number of readers to participate in the service while allowing operations to complete in one round. In contrast to the classic approach of the two rounds per read operation, OhFast tries to further reduce the communication required by *slow* reads. Thus OhFast combines ideas from ccFast and the

Algorithm 2 Read protocol of algorithm OhFast.

```
1: at each reader r_i
2: Components:
3: ts ∈ ℕ⁺; maxTS ∈ ℕ⁺; v, vp ∈ V; rcounter ∈ ℕ⁺
4: srvAck ⊆ S × M
5: Initialization:
6: ts ← 0, maxTS ← 0, v ← ⊥, vp ← ⊥; rcounter ← 0
7: function READ( )
8:   rcounter ← rcounter + 1
9:   send(⟨ts, v, vp⟩, r_i, rcounter) to all servers
10:  wait until |S| − f servers reply
     ▷ Collect the (sid, ⟨⟨ts′, v′, vp′⟩, views, secured⟩) msgs in srvAck
11:  maxTS ← max{m.ts′|(s, m) ∈ srvAck}
12:  maxAck ← {(s, m)|(s, m) ∈ srvAck ∧ m.ts′ = maxTS}
13:  ⟨ts, v, vp⟩ ← m.⟨ts′, v′, vp′⟩ for (∗, m) ∈ maxAck
14:  maxViews ← max{m.views|(s, m) ∈ maxAck}
15:  if ∃(s, m) ∈ maxAck s.t. m.secured = True then
16:    return(v)
17:  else if ∃α ∈ [1, |S|/f − 2] s.t.
18:    MS = {s : (s, m) ∈ maxAck ∧ m.views ≥ α} and
19:    |MS| ≥ |S| − αf then
20:    return(v)
21:  else
22:    return(vp)
23:  end if
24: end function
```

one and a half round approach suggested by OhSam. With server to server communication, OhFast is expected to perform better in environments where the servers communicate via high capacity links, e.g., data centers.

Like in OhSam, servers assume the responsibility of propagating the value of the timestamp instead of the reader. Similarly, in OhFast we move the decision on a slow read to the servers. In particular, the servers record the processes that requested their timestamp. If the recording set becomes "large" then a server relays a read to the other servers before replying to the reader. However, there is a major departure from OhSam: the servers that receive relay messages do not broadcast relays to all the servers but just to the servers that send them a relay. So, only a single server may relay for a read operation keeping the message complexity of the algorithm low in cases of low contention. When a server that relays a timestamp gets appropriate relays from the other servers, it marks the timestamp as *secured*, and sends a reply to the reader. When now the reader receives the replies from $|\mathcal{S}| - f$ servers it collects the messages with the highest timestamp. If there is a server that declares this timestamp as *secured* then the read immediately returns the value associated with this timestamp; otherwise the reader evaluates the predicate of ccFast on the replies to determine the value to return.

Algorithms 2 and 3 provide the formal pseudocode of OhFast. We omit the write protocol as it is the same to the one presented for ccHybrid. The read protocol in OhFast (Algorithm 2) is simpler than the read of ccHybrid. The reader sends messages to all the servers and waits for $|\mathcal{S}| - f$ of them to reply (L9). Once those replies are received the reader discovers the maximum timestamp $maxTS$ among the replies (L11), and collects the messages that contain $maxTS$ (L12) [1] in the set $maxAck$. If some message in $maxAck$ indicates that $maxTS$ is secured, i.e. it contains $secured = True$ (L15), then the reader returns $maxTS$. Otherwise, it evaluates the predicate on the messages in $maxAck$ (L19) to determine which timestamp to return.

The server protocol (Algorithm 3) is the most involved in OhFast. The server's state is composed of the state of the replica, the recording set *seen*, a flag *securedts* which indicates whether a timestamp has been relayed to a majority of servers, and a *Relays* list storing the latest timestamp the server relayed for each reader. A server s waits for read/write and relay requests. When s receives a read/write request it updates its local replica state and *seen* set appropriately (L13-14). In case the timestamp in the request is higher than its local timestamp it also sets *securedts* flag to $False$. Then, s decides whether to relay the received timestamp or not. In particular, s relays a timestamp if (L19):

[1] Notice that this is another departure from OhSam as each reader in OhSam returns the smallest discovered timestamp.

Algorithm 3 Server protocol of algorithm OhFast.

```
1: at each server s_j
2: Components:
3: ts ∈ ℕ+; seen ⊆ ℛ ∪ {w}; v, vp ∈ V; Counter[|ℛ|+1] ∈ ℕ+
4: scounter ∈ ℕ+; securedts ∈ {True, False}
5: Relays[|ℛ|] ∈ ℕ+
6: Initialization:
7: ts ← 0; seen ← ∅; v, vp ← ⊥; prop ← False
8: Counter[i] ← 0 for i ∈ ℛ ∪ {w}; scounter ← 0
9: Relays[i] ← 0; securedts ← False
10: function rcv(⟨ts', v', vp'⟩, q, counter)
        ▷ Called upon reception of a READ/WRITE message
11:   if Counter[q] < counter then
12:     if ts' > ts then
13:       ⟨ts, v, vp⟩ ← ⟨ts', v', vp'⟩; seen ← {q}
14:       securedts ← False
15:     else
16:       seen ← seen ∪ {q}
17:     end if
18:     if q ∈ ℛ and |seen| > |𝒮|/f − 2 and
19:       securedts = False and Relays[q] < ts then
20:       scounter ← scounter + 1
21:       sendRelay(⟨ts, v, vp⟩, q, s_j, counter, scounter)
22:         to all the servers
23:       Relays[q] ← ts; srvRelay ← ∅
24:     else
25:       send(⟨ts, v, vp⟩, |seen|, counter, securedts) to q
26:     end if
27:   end if
28: end function
29: function rcvRelay(⟨ts', v', vp'⟩, q, s, c1, c2)
        ▷ Called upon reception of a RELAY message
30:   if Counter[s] < c2 then
31:     if ts' > ts then
32:       ⟨ts, v, vp⟩ ← ⟨ts', v', vp'⟩
33:       seen ← {q}
34:     else if ts = ts' then
35:       seen ← seen ∪ {q}
36:     end if
37:     if Relays[q] = ts' then
38:       srvRelay ← srvRelay ∪ {s}
39:       if |srvRelay| = |𝒮| − f then
40:         if ts = ts' then
41:           securedts ← True
42:         end if
43:         send(⟨ts', v', vp'⟩, 0, c1, True) to q
44:       end if
45:     else
                                  ▷ reply back to the sender
46:       scounter ← scounter + 1
47:       sendRelay(⟨ts', v', vp'⟩, q, s_j, scounter) to s
48:     end if
49:   end if
50: end function
```

(i) the sender is a reader,
(ii) it sent this timestamp to more than $\frac{|\mathcal{S}|}{f} - 2$ processes,
(iii) the timestamp has not already being relayed (i.e. $securedts = False$) and
(iv) the server has not yet relayed this timestamp for the same reader.

If some of these conditions does not hold then s just replies to the sender with its local timestamp (L25). Notice here that servers only relay for the readers and do not relay for the writer, as the sole writer always has the latest timestamp. In a relay message s includes its local replica state, the id of the reader that initiated the relay, and its own id. When a server s' receives a relay message from s, it first updates its local replica and *seen* set appropriately (L32-33, L35). Then s' checks if it also sent a relay with the same timestamp for the same reader (L37). If not then s' bounces the relay to s and completes (L47); otherwise s' adds s in the servers that received its relay 38). When it receives $|\mathcal{S}| - f$ relays, s' replies to the reader that initiated the relay along with the timestamp that it initially relayed (not its local timestamp) (L43). Finally, if its local timestamp is the same as the relayed timestamp, then s' also sets $securedts = True$ (L41).

5.1 Algorithm Correctness

In order to show that OhFast is correct we have to prove that it satisfies both termination (liveness) and atomicity (safety) properties. Termination of the write operation is easy to see as according to our failure model $|\mathcal{S}| - f$ servers do not fail and can receive and reply to the write request. However, termination of the read protocol is not straightforward: a server may communicate with other servers before responding to a reader. The next lemma shows that all the read operations terminate.

▶ **Lemma 6.** *In any execution ξ of* OhFast*, every read operation ρ invoked by a correct process r eventually terminates.*

Next it remains to show that atomicity is preserved. To prove atomicity we are going to use the four properties that express atomicity in terms of timestamps written and returned, as presented in Section 4.1. It is easy to see from the algorithm, that every process updates its local replica only when a value with a higher timestamp is received. Thus, it can be easily seen that the algorithm satisfies properties **A1** and **A3**. Notice also that when a server

receives a timestamp ts then it attaches a timestamp $ts_s \geq ts$ to any message it sends from that point onward. This can be shown with similar statements as in Lemma 1. We need to show that when a server receives a *relay* that contains a timestamp ts then it sends a timestamp $ts_s \geq ts$ from that point onward.

▶ **Lemma 7.** *In any execution ξ of* OhFast*, if a server s receives a relay with a timestamp ts at time T from a server s', then s attaches a timestamp $ts' \geq ts$ to any message it sends at any time $T' > T$.*

Now we can show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written. This shows the validity of property **A2**.

▶ **Lemma 8.** *In any execution ξ of the algorithm, if a read ρ from r succeeds a write operation ω that writes timestamp ts_ω from the writer w, i.e. $\omega \to \rho$, and returns a timestamp ts_ρ, then $ts_\rho \geq ts_\omega$.*

Finally, it remains to investigate if property **A4** holds. Before we do so, we prove a lemma showing that if a timestamp ts is secured from a server s, then at least $|\mathcal{S}| - f$ servers have a timestamp $ts' > ts$.

▶ **Lemma 9.** *In any execution ξ of* OhFast*, if a server s sets $securedts = True$ for a timestamp ts at time T then $\exists \mathcal{S}' \subseteq \mathcal{S}$ at T, s.t. $|\mathcal{S}'| \geq |\mathcal{S}| - f$ and $\forall s' \in \mathcal{S}'$, the local timestamp of s' is $ts' \geq ts$.*

▶ **Lemma 10.** *In any execution ξ of* OhFast*, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$, and ρ_1 returns ts_{ρ_1}, then ρ_2 returns $ts_{\rho_2} \geq ts_{\rho_1}$.*

Proof. A read operation may decide on the value to return in two ways in OhFast: (i) it receives a secured timestamp, or (ii) it evaluates the predicate. Let us first examine what happens when the two reads are invoked by the same reader (i.e. $r_1 = r_2$). During ρ_2, r_1 includes a timestamp $ts_{r_1} \geq ts_{\rho_1}$ in every message it sends to servers. According to Lemma 1 every server s replies with a timestamp $ts_s \geq ts_{\rho_1}$. Thus, $maxTS_2 \geq ts_{\rho_1}$. If $maxTS_2 > ts_{\rho_1}$ then since $ts_{\rho_2} = maxTS_2$ or $ts_{\rho_2} = maxTS_2 - 1$ it follows that $ts_{\rho_2} \geq ts_{\rho_1}$ in either case. If $maxTS_2 = ts_{\rho_1}$ then every server adds r_1 in their *seen* set before replying to ρ_2. So the predicate is valid for $|MS| \geq |\mathcal{S}| - f$ and $\alpha = 1$. Hence, ρ_2 returns $ts_{\rho_2} = maxTS_2 = ts_{\rho_1}$ in any case (i) or (ii).

So we need now to examine all the possible combinations for the two reads ρ_1 and ρ_2 when $r_1 \neq r_2$. If both read operations examine the predicate to decide on the value to return (i.e., they do not receive a secured timestamp), then with same reasoning as in [3, Lemma 8] we can show that atomicity is preserved. So it remains to examine the following three cases:

1. ρ_1 evaluates the predicate, and ρ_2 receives a secured $maxTS_2$,
2. ρ_1 receives a secured $maxTS_1$, and ρ_2 evaluates the predicate, and
3. ρ_1 receives a secured $maxTS_1$, and ρ_2 receives a secured $maxTS_2$.

Case 1: In this case, ρ_1 evaluates the predicate, and ρ_2 returns $ts_{\rho_2} = maxTS_2$ as it received a reply with $maxTS_2$ and $secured = True$. There are two subcases to examine: (a) ρ_1 returns $maxTS_1$, and (b) ρ_1 returns $maxTS_1 - 1$.

Case 1a: If ρ_1 returns $maxTS_1$ it follows that the predicate is valid for ρ_1. Hence:

$$\exists \alpha \in [1, \frac{|\mathcal{S}|}{f} - 2] \text{ and}$$
$$MS \subseteq \mathcal{S} \text{ s.t. } MS = \{s : s.ts = maxTS_1 \ \wedge \ s.views \geq \alpha\} \ \wedge \ |MS| \geq |\mathcal{S}| - \alpha f \, .$$

Moreover, since ρ_1 examines the predicate, then none of the servers that replied with $maxTS_1$ sends $secured = True$. Therefore, $\forall s \in MS$, it must be true that $s.views \leq \frac{\mathcal{S}}{f} - 2$ before replying to ρ_1 (L16), otherwise s would proceed to relay and secure $maxTS_1$. Since every $s.views \leq \frac{\mathcal{S}}{f} - 2$, then it must be the case that $\alpha \leq \frac{\mathcal{S}}{f} - 2$ as well. Thus substituting:

$$|MS| \geq |\mathcal{S}| - \alpha f \Rightarrow |MS| \geq |\mathcal{S}| - (\frac{\mathcal{S}}{f} - 2)f \Rightarrow |MS| > f\,.$$

Since ρ_2 receives replies from $|\mathcal{S}_2| = |\mathcal{S}| - f$ servers then $\mathcal{S}_2 \cap MS \neq \emptyset$. Also notice that since $\rho_1 \rightarrow \rho_2$, then a server $s \in \mathcal{S}_2 \cap MS$ replies to ρ_1 with $maxTS_1$ before replying to ρ_2. By Lemma 1, s replies to ρ_2 with a timestamp $ts_s \geq maxTS_1$. Thus, $maxTS_2 \geq ts_s \Rightarrow maxTS_2 \geq maxTS_1$ and ρ_2 returns $ts_{\rho_2} \geq maxTS_1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$.

Case 1b: Assume now the case where ρ_1 returns $maxTS_1 - 1$. Since ρ_1 received $maxTS_1$, and since the sole writer invokes one operation at a time, then it follows that the write operation that wrote $maxTS_1 - 1$, say ω, completed during or before ρ_1. Since though $\rho_1 \rightarrow \rho_2$, then it follows that $\omega \rightarrow \rho_2$. Since ω communicates with $|\mathcal{S}| - f$ servers before completing, and since ρ_2 waits for $|\mathcal{S}| - f$ replies, then there is a server s that replies to ω before replying to ρ_2. By Lemma 1, s replies with a timestamp $ts_s \geq maxTS_1 - 1$ to ρ_2. Thus ρ_2 observes a $maxTS_2 \geq maxTS_1 - 1$, and hence $ts_{\rho_2} \geq maxTS_1 - 1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$ in this case as well.

Case 2: Here, ρ_1 returns $ts_{\rho_1} = maxTS_1$ as it received a message that contained $maxTS_1$ and $secured = True$. Read ρ_2 evaluates the predicate to decide on the value to return. We have two subcases to examine again: (a) ρ_2 returns $maxTS_2$, or (b) ρ_2 returns $maxTS_2 - 1$. Since ρ_1 returned a secured timestamp, then it received $maxTS_1$ and $secured = True$ from some server s. By Lemma 9, a set $|\mathcal{S}'| \geq |\mathcal{S}| - f$ of servers have a timestamp $ts' \geq maxTS_1$ before s replies to ρ_1. Since ρ_2 receives replies from $|\mathcal{S}_2| = |\mathcal{S}| - f$ servers, then $\mathcal{S}' \cap \mathcal{S}_2 \neq \emptyset$. Then by Lemmas 1 and 7, any server in $s' \in \mathcal{S}' \cap \mathcal{S}_2$ replies to ρ_2 with a timestamp $ts_{s'} \geq maxTS_1$. Thus, ρ_2 observes a $maxTS_2 \geq maxTS_1$. If $maxTS_2 > maxTS_1$ and since ρ_2 returns either $maxTS_2$ or $maxTS_2 - 1$, then in either case $ts_{\rho_2} \geq ts_{\rho_1}$.

So it remains to examine what happens when $maxTS_2 = maxTS_1$. If ρ_2 returns $ts_{\rho_2} = maxTS_2$ then $ts_{\rho_2} \geq ts_{\rho_1}$. Let us examine now if ρ_2 may return $maxTS_2 - 1$. As we said before every server s' in $\mathcal{S}' \cap \mathcal{S}_2$ replies with $ts_{s'} \geq maxTS_1$ to ρ_2. Since $|\mathcal{S}'| \geq |\mathcal{S}| - f$ and $|\mathcal{S}_2| \geq |\mathcal{S}| - f$ then $|\mathcal{S}' \cap \mathcal{S}_2| \geq |\mathcal{S}| - 2f$. Also by the algorithm, every server in $\mathcal{S}'$ adds r_1 in its *seen* set before replying to the relay message from s (L39). Furthermore, every server in $\mathcal{S}_2$ adds r_2 in its *seen* set before replying to ρ_2. So every server $s' \in \mathcal{S}' \cap \mathcal{S}_2$ replies with a $s.views \geq 2$. Thus, the predicate holds for at least $|MS| = |\mathcal{S}' \cap \mathcal{S}_2| \geq |\mathcal{S}| - 2f$ and $\alpha = 2$. Hence ρ_2 will return $maxTS_2$ contradicting our assumption that returns $maxTS_2 - 1$. So returning $maxTS_2 - 1$ is not possible.

Case 3: In this case both ρ_1 and ρ_2 return a secured timestamp. Let s_1 be the server that send $maxTS_1$ and $secured = True$ to ρ_1, and s_2 (not necessarily different than s_1) be the server that sent $maxTS_2$ and $secured = True$ to ρ_2. By Lemma 9, there exists a set $\mathcal{S}'$ s.t. every server $s \in \mathcal{S}'$ has a timestamp $ts_s \geq maxTS_1$ before s_1 replies to ρ_1. As explained in Case 2, $\mathcal{S}' \cap \mathcal{S}_2 \neq \emptyset$. Hence there exists a server that replied both to the relay message of s_1 and to ρ_2. By Lemma 7, each server $s' \in \mathcal{S}' \cap \mathcal{S}_2$ replies to ρ_2 with a timestamp $ts_{s'} \geq maxTS_1$. Hence, $maxTS_2 \geq maxTS_1$. Since ρ_2 returns a secured timestamp, then it returns $maxTS_2$. Therefore, $ts_{\rho_2} = maxTS_2 \Rightarrow ts_{\rho_2} \geq maxTS_1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$. ◀

▶ **Theorem 11.** *Algorithm* OhFast *implements a SWMR atomic read/write register.*

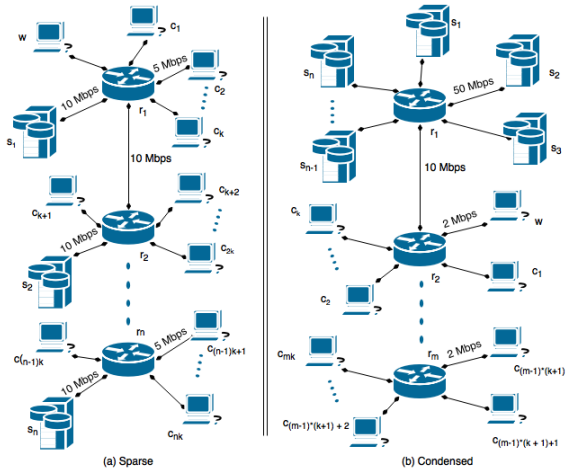


Figure 1 Simulated topologies.

6 Empirical Results

In this section, we present empirical results that we obtained by implementing algorithms ABD [1], OhSam [6], Sf [5], ccHybrid, and OhFast, using the NS3 discrete event simulator [11]. NS3 is a highly customizable and extensible simulator that allows us to gain full control over the event scheduler and the deployment environment. Thus, it allows us to investigate the exact parameters that may affect the performance of our algorithms.

Experimentation Platform. The general testbed of our experiments consists of a single writer, a set of readers, and a set servers. We assume that $f = 1$ servers may fail. This assumption was chosen so as every operation would wait for all but one servers to reply, inflicting that way high concurrency and potentially inconsistency in our system. Communication between the nodes is established via point to point bidirectional links implemented with a DropTail queue. For the purpose of the experimental evaluation, we developed simulations representing two different topologies, *Sparse* and *Condensed*, which mainly differ on the deployment of server nodes.

Figure 1 presents the two topologies. In both topologies the clients are divided evenly and are connected on a series of router nodes. Clients are connected to the routers with 5Mbps links and 2ms delay, and routers are connected with 10Mpbs links and 4ms delay. In the *Sparse* topology (Figure 1(a)), a server is connected to each router with 10Mbps bandwidth and 2ms delay. This topology demonstrates a network where servers are separated and appear to be in different networks. In the *Condensed* topology (Figure 1(b)) all the servers are connected to a single router with 50Mbps links and 2ms delay, simulating a network where servers are connected in close proximity and with high bandwidth links (e.g., a datacenter).

We ran NS3 on a Macintosh machine running OS X El Capitan, with 2.5Ghz Intel Core i7 processor and 16GB of RAM. The average of 5 samples per scenario provided the stated operation latencies.

Performance. The performance of the algorithms is measured in terms of the ratio of the number of fast over slow R/W operations – *communication burden*; and the total time it takes for an operation to complete – *operation latency*. Operation latency is affected by both

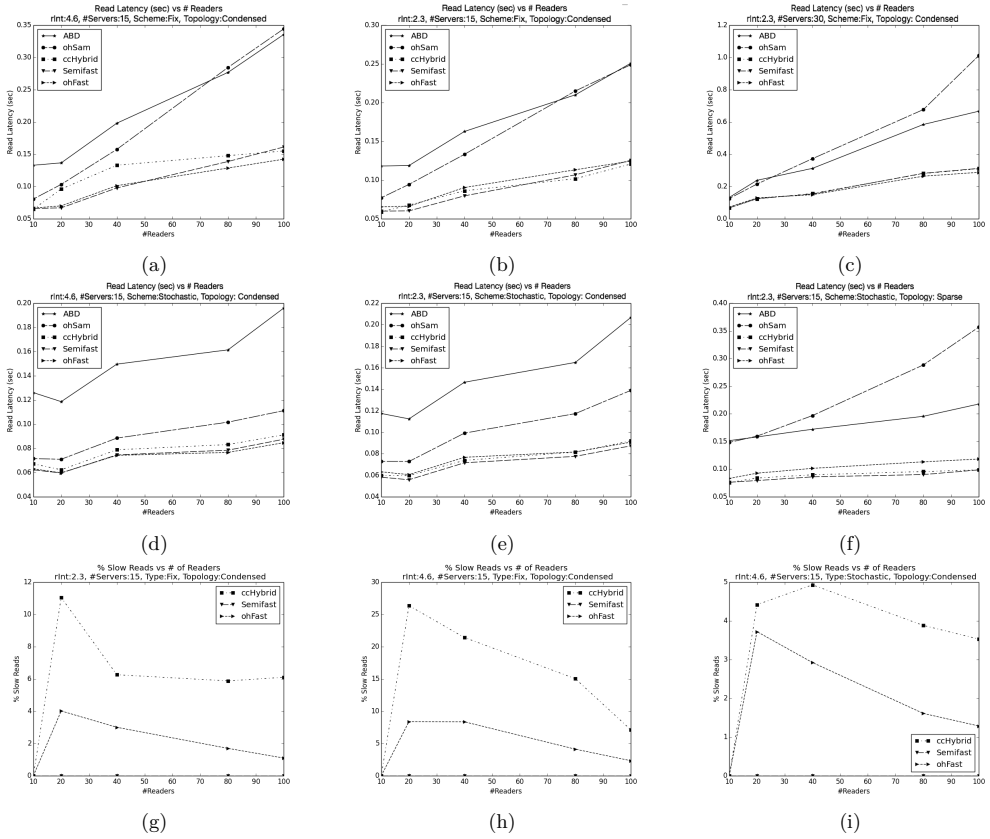


Figure 2 Experimental Results from NS3 Simulation.

communication and computation latencies. As NS3 only provides simulated time events and omits any computation, we combined two clocks: (a) the simulation clock, and (b) a real time clock. The simulation clock was able to estimate the communication time, while the real clock allowed us obtain the time taken by the computation at each operation. The latency is calculated adding both times.

Scenarios. Measurements of the performance involves multiple execution scenarios. The scenarios were designed to test

(i) the scalability of the algorithms as the number of readers and servers increases;
(ii) the contention effect on efficiency, by running different concurrency scenarios; and
(iii) the relation of the efficiency with the topology of the network that we use.

To test scalability we range the number of readers $|\mathcal{R}| \in [10, 20, 40, 80, 100]$ and the number of servers $|\mathcal{S}| \in [10, 15, 20, 25, 30]$. To test contention we specify the frequency of read operation and we run our algorithm for different read intervals ($rInt \in [2.3, 4.6, 6.9]$ seconds). We issue write operations every 4 seconds. In addition, we define two read invocation schemes: (i) *fix* and (ii) *stochastic*. In the fix scheme all the read operations are scheduled periodically at the read interval. In the stochastic scheme each operation is scheduled at random between $1s$ and $rInt$ seconds in each read interval. Finally, to test topological effects we run our algorithms using both the *Sparse* and *Condensed* topologies.

Results

As a general observation, the new algorithms outperform all the other algorithms in most scenarios. In particular, it is clear that ccHybrid and OhFast outperform algorithms ABD and OhSam. In addition, the two algorithms appear to achieve similar operation latencies as Sf. A closer examination reveals that in many scenarios Sf does not perform any slow reads, whereas in the same executions both ccHybrid and OhFast require some slow reads. The fact that the two algorithms perform the same as Sf, despite the slow reads, demonstrates that the computation overhead of the two presented algorithms is much less than the computation needed by Sf. Thus, in executions where Sf will perform more slow operations, clearly this will result in even worse operation latencies. More in detail, taking our tests one by one we conclude to the following observations:

Scalability: As can be seen in Figures 2(b) and (c), the increasing number of readers and the servers have a negative impact on all the algorithms. The impact is higher on ABD and OhSam, and lower for the rest of the algorithms.

Contention: Contention is generated by:
(i) operation frequencies, and
(ii) concurrency schemes.
We observe that *operation frequency* affects the latency of the operations in the *fix* scheme. This can be seen in Figure 2(a) and (b). Algorithms ABD and OhSam are not affected (as all of their reads are slow), but the multi-speed algorithms Sf, ccHybrid and OhFast, are affected negatively. This behaviour is due to the fact that these algorithms perform a slow read operation per write operation. When the read interval is close to the write interval, e.g., $rInt = 4.6$, most of the reads are concurrent to the write and thus more reads are slow (Figure 2(h)). This is not the case when $rInt = 2.3$ (Figure 2(g)). Notice that the same behavior is not being observed when a *stochastic* scheme is used, as radomness prevents the operations to be invoked at exactly the same time (Figure 2(d) and (e)). Hence, a slow read operation may complete before any read operations that return the same value are invoked. Therefore, according to the multi-speed algorithms, once a slow read is completed, any read operation that succeeds such a read will be fast. This results in a low percentage of slow reads, as shown in Figure 2(i).

Finally, when the operation frequency is constant, it appears that in the *stochastic* scheme each operation completes almost two times faster than in the *fix* scheme (Figure 2(b) and 2(e)). Algorithms, ABD and OhSam, can be used as points of reference as they have the same computation and communication requirements in both *fix* and *stochastic* scenarios. The difference can be explained due to the congestion that the *fix* scheme introduces in the network. On the contrary, a *stochastic* scheme distributes the invocation time intervals of the read operations uniformly, reducing the network congestion, and hence operation latency.

Topology: Plots 2(e) and 2(f) show that topology has an impact on the performance and the efficiency of all the algorithms. Most importantly, we can observe that OhSam and OhFast are the two algorithms that are affected the most. In particular, while in (e) OhSam performs better than ABD and OhFast performs similar to ccHybrid and Sf we notice that in (f) OhSam performs worse than ABD and OhFast worse than the 2 others. This behaviour is expected as both OhSam and OhFast need to exchange messages between the servers during a relay phase. However, notice that OhFast performs much better since operation relays are not performed for every read operation.

7 Conclusions

In this paper we present two new algorithms ccHybrid and OhFast that implement atomic SWMR register in a message-passing, asynchronous environment. Both algorithms use the predicate introduced in [3], to achieve *single round* reads with small computational footprint. However, to avoid constraints in reader participation both algorithms allow some reads to be *slow*. In ccHybrid the reader decides on the speed of its read operation, reasulting in operations that perform *1 or 2 rounds*. OhFast moves the decision of slow operations to the servers, enabling *1 or* 1.5 *round* operations. Simulation results show that our algorithms outperform all slow operation algorithms, as well as "multi-speed" implementations that have high computation demands. We claim that our developments take us closer to *practical* implementations of atomic read/write objects in the message-passing environment.

References

1 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.

2 P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC04)*, pages 236–245, 2004.

3 A. Fernández Anta, N. Nicolaou, and A. Popa. Making "fast" atomic operations computationally tractable. In *Proceedings 19th International Conference On Principle Of DIstributed Systems (OPODIS 15)*, 2015.

4 Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC'08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 289–304, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87779-0_20.

5 Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009. doi:10.1016/j.jpdc.2008.05.004.

6 T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann. Brief announcement: Oh-ram! one and a half round read/write atomic memory. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC'16, pages 353–355, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933073.

7 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

8 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Transactions on Computers*, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

9 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

10 Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

11 NS3 network simulator. URL: https://www.nsnam.org/.

Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Register Emulations*

Tiago Oliveira[1], Ricardo Mendes[2], and Alysson Bessani[3]

1 **LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal**
`toliveira@lasige.di.fc.ul.pt`
2 **LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal**
`rmendes@lasige.di.fc.ul.pt`
3 **LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal**
`anbessani@ciencias.ulisboa.pt`

Abstract

Resilient register emulation is a fundamental technique to implement dependable storage and distributed systems. In data-centric models, where servers are modeled as fail-prone base objects, classical solutions achieve resilience by using fault-tolerant quorums of read-write registers or read-modify-write objects. Recently, this model has attracted renewed interest due to the popularity of cloud storage providers (e.g., Amazon S3), that can be modeled as key-value stores (KVSs) and combined for providing secure and dependable multi-cloud storage services. In this paper we present three novel wait-free multi-writer multi-reader regular register emulations on top of Byzantine-prone KVSs. We implemented and evaluated these constructions using five existing cloud storage services and show that their performance matches or surpasses existing data-centric register emulations.

1998 ACM Subject Classification D.4.2 Storage Management

Keywords and phrases Byzantine fault tolerance, register emulation, multi-writer, key-value store, data-centric algorithms

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.30

1 Introduction

Resilient register emulations on top of message passing systems are a cornerstone of fault-tolerant storage services. These emulations consider the provision of shared objects supporting read and write operations executed by a set of clients. In the traditional approach, these objects are implemented in a set of fail-prone servers (or replicas) that run some specific code for the emulation [4, 25, 19, 33, 14, 27, 9, 17, 26].

A less explored approach, dubbed *data-centric*, does not rely on servers that can run arbitrary code, but on passive replicas modeled as base objects that provide a constrained interface. These base objects can be as simple as a network-attached disk, a remote addressable memory, or a queue, or as complex as a transactional database, or a full-fledged cloud storage service. By combining these fail-prone base objects, one can build fault-tolerant services for storage, consensus, mutual exclusion, etc, using only *client-side code*, leading to arguably simpler and more manageable solutions.

* This work was supported by FCT through projects LaSIGE (UID/CEC /00408/2013) and IRCoC (PTDC/EEI-SCR/6970/2014), and by EU through the H2020 SUPERCLOUD project (643964).

The data-centric model has been discussed since the 90s [21], but the area gained visibility and practical appeal only with the emergence of network-attached disks technology [16]. In particular, several theoretical papers tried to establish lower bounds and impossibility results for implementing resilient read-write registers and consensus objects considering different types of fail-prone base objects (read-write registers [2, 15] vs. read-modify-write objects [12, 11]) under both crash and Byzantine fault models [1]. More recently, there has been a renewed interest in data-centric algorithms for the cloud-of-clouds model [6]. In this model, the base objects are cloud services (e.g., Amazon S3, Windows Azure Blob Storage) that offer interfaces similar to read-write registers or *key-value stores* (KVSs). This approach ensures that the stored data is available even if a subset of cloud providers is unavailable or corrupts their copy of the data (events that do occur in practice [24]).

To the best of our knowledge, there are only two existing works for register emulation in the cloud-of-clouds model: DepSky [6], which tolerates Byzantine faults (e.g., data corruption or cloud misbehavior) on the providers but *supports only a single-writer per data object*, and Basescu et al. [5], which genuinely supports multiple writers, but *tolerates only crash faults and does not support erasure codes.*

In this paper we present new register emulations on top of cloud storage services that support multiple concurrent writers (avoiding the need for expensive mutual exclusion algorithms [6]), tolerate Byzantine failures in base objects (minimizing the trust assumptions on cloud providers), and integrate erasure codes (decreasing the storage requirements significantly). In particular, we present three new multi-writer multi-reader (MWMR) regular register constructions:

1. an optimally-resilient register using full replication;
2. a register construction requiring more base objects, but achieving better storage-efficiency through the use of erasure codes;
3. another optimally-resilient register emulation that also supports erasure codes, but requires additional communication steps for writing.

These constructions are wait-free (operations terminate independently of other clients), uniform (they work with any number of clients), and can be adapted to provide atomic (instead of regular) semantics.

We achieve these results by exploring an often overlooked operation offered by KVSs – *list* – which returns the set of stored keys. The basic idea is that by embedding data integrity and authenticity proofs in the key associated with a written value, it is possible to use the list operation in multiple KVSs to detect concurrent writers and establish the current value of a register. Although KVSs are equivalent to registers in terms of synchronization power [8], the existence of the list operation in the interface of the former is crucial for our algorithms.

Besides the reduction on the storage requirements, an additional benefit of supporting erasure codes when untrusted cloud providers are considered is that they can be substituted by a secret sharing primitive (e.g., [22]) or any privacy-aware encoding (e.g., [9, 30]), ensuring confidentiality of the stored data.

The three constructions we propose are described, proved correct, implemented and evaluated using real clouds (Amazon S3 [3], Microsoft Azure Storage [10], Rackspace Cloud Files [31], Google Storage [18] and Softlayer Cloud Storage [32]). Our experimental results show that these novel constructions provide advantages both in terms of latency and storage costs.

Table 1 Data-centric resilient register emulations. *: Can be extended to achieve atomic semantics.

Work	Fault model	Technique	Base Objects	Resilience	Semantics
Jayanti et al. [21]	Byzantine	replication	atomic registers	$5f+1$	SW safe
Gafni and Lamport [15]	crash	replication	atomic registers	$2f+1$	SW regular
Chockler and Malkhi [12]	crash	replication	rmw registers	$2f+1$	MW ranked
Abraham et al. [1]	Byzantine	replication	regular registers	$3f+1$	SW regular
	Byzantine	replication	regular registers	$3f+1$	SW safe
Aguilera and Gafni [2]	crash	replication	atomic registers	$2f+1$	MW atomic
Bessani et al. [6]	Byzantine	replication	regular registers	$3f+1$	SW regular
	Byzantine	erasure code	regular registers	$3f+1$	SW regular
Basescu et al. [5]	crash	replication	atomic KVSs	$2f+1$	MW regular*
This paper	Byzantine	replication	atomic KVSs	$3f+1$	MW regular*
	Byzantine	erasure code	atomic KVSs	$4f+1$	MW regular*
	Byzantine	erasure code	atomic KVSs	$3f+1$	MW regular*

2 Related Work

Existing fault-tolerant register emulations can be divided in two main groups depending on the nature of the fail-prone "storage blocks" that keep the stored data. The first group comprises the works that rely on servers capable of running part of the protocols [4, 25, 19, 33, 14], i.e., constructions that have both a client-side and a server-side of the protocol. Typically, in this kind of environment it is easier to provide robust solutions as servers can execute specific steps of the protocol atomically, independently of the number of clients accessing it.

In the second group we have the *data-centric* protocols [21, 2, 15, 12, 1]. This approach considers a set of clients interacting with a set of passive servers with a constrained interface, modeled as shared memory objects (called base objects). The first work in this area was due to Jayanti, Chandra and Toueg [21], where the model was defined in terms of fail-prone shared memory objects. This work presented, among other wait-free emulations [20], a Byzantine fault-tolerant single-writer single-reader (SWSR) safe-register construction using $5f+1$ base objects to tolerate f faults. Further works tried to establish lower bounds and impossibility results for emulating registers tolerating different kinds of faults considering different types of base objects. For example, Aguilera and Gafni [2] and Gafni and Lamport [15] used regular and/or atomic registers to implement crash-fault-tolerant MW and SW registers,[1] respectively, while Chockler and Malkhi [12] used read-modify-write objects to transform the SW register of Gafni and Lamport [15] in a ranked register, a fundamental abstraction for implementing consensus. Abraham et al. [1] provided a Byzantine fault-tolerant SW register, which was latter used as a basis to implement consensus. The main limitation of these algorithms is that, although they are asymptotically efficient [2], the number of communication steps is still very large, and the required base objects are sometimes stronger than KVSs [12] or implement weak termination conditions [1].

More recently, there has been a renewed interest in data-centric algorithms for the cloud-of-clouds model [6, 5]. Here the base objects are cloud services offering interfaces similar to key-value stores. These solutions ensure that the stored data is available even if a subset of cloud providers is unavailable or corrupts their copy of the data. DepSky [6]

[1] From now on we avoid characterizing the constructions about the number of readers, as all constructions discussed in the rest of the paper support multiple readers (MR).

provided a regular SW register construction that tolerates Byzantine faults by less than a third of the base objects, ensuring also the confidentiality of the stored data by using a secret sharing scheme [22]. However, to support multiple writers an expensive lock protocol must be executed to coordinate concurrent accesses. Another work in this line [5] provided a regular MW register that replicates the data by a majority of KVSs. Its main purpose was to reduce the necessary storage requirements. To achieve that, writers remove obsolete data synchronously, creating the need to store each version in two keys: a temporary key, that could be removed, and an eternal key, common for all writers and versions, that is never erased. In the best case, the algorithm requires a storage space of $2 \times S \times n$, where S is the size of the data and n is the number of KVSs.

Using registers or KVSs as base objects in the data-centric model makes it more challenging to implement dependable register emulations, as general replicas have more synchronization power than such objects [8]. The three new register constructions presented in this paper advance the state of the art by supporting multiple writers and erasure-coded data in the data-centric Byzantine model, using a rather weak base object – a KVS. Two of these constructions have optimal resilience, as they require $3f + 1$ base objects to tolerate f Byzantine faults in an asynchronous system (with confirmable writes) [27]. Table 1 summarizes the discussed data-centric constructions.

3 System Model

3.1 Register Emulation

We consider an *asynchronous* system composed of a finite set of clients and n cloud storage providers that provide a KVS interface. We refer to clients as *processes* and to cloud storage providers as *base objects*. Each process has a unique identifier from an infinite set named *IDs*, while the base objects are numbered from 0 to $n - 1$.

We aim to provide *MW-register* abstractions on top of n base objects. Concretely, a register abstraction offers an interface specification composed of two *operations*: **write(v)** and **read()**. The sequential specification of a register requires that a read operation returns the last value written, or $\perp$ if no write has ever been executed. Processes interacting with registers can be either *writers* or *readers*.

A process operation starts by an *invoke* action on the register, and ends with a *response*. An operation *completes* when the process receives the response. An operation o_1 *precedes* another operation o_2 (and o_2 *follows* o_1) if it completes before the invocation of o_2. Operations with no precedence relation, are called *concurrent*.

Unless stated otherwise, the register implementations should be *wait-free* [20], i.e., the operation invocations should complete in a finite number of internal steps. Moreover, we provide *uniform* implementations, i.e., implementations that do not rely on the number of processes, allowing processes to not know each other initially.

We provide two register abstraction semantics, *regular* and *atomic*, which differ mainly in the way they deal with concurrent accesses [23]. A regular register guarantees only that different read operations agree on the order of preceding write operations. Any read operation overlapping a write operation may return the value being written or the preceding value. An atomic register employs a stronger consistency notion than regular semantics. It stipulates that it should be possible to place each operation at a singular point (linearization point) between its invocation and response. This mean that after a read operation completes, a following read must return at least the version returned in the preceding read, even in the presence of concurrent writes.

3.2 Threat Model

Up to f out-of n base objects can be subject to NR-arbitrary failures [21], which are also known as Byzantine failures. The behavior of such objects can be unrestricted: they may not respond to an invocation, and if they do, the content of the response may be arbitrary. Unless stated otherwise, readers may also be subject to Byzantine failures. Writers can only fail by crashing, because even if the protocol tolerates Byzantine writers, they may always store arbitrary values or overwrite data on the register. Processes and base objects are said to be *correct* if they do not fail.

For cryptography, we assume that each writer has a private key K_r to sign some of the information stored on the base objects. These signatures can be verified by any process in the system through the corresponding public key K_u. Moreover, we also assume the existence of a collision-resistant cryptographic hash function to ensure integrity. There might be multiple writer keys as long as readers can access their public counterparts.

3.3 Key-Value Store Specification

Current cloud storage service providers offer a key-value store (KVS) interface, which act as a passive server where it is impossible to run any code, forcing the implementations to be *data-centric*. Specifically, KVSs allow customers to interact with associative arrays, i.e, with a collection of $\langle key, value \rangle$ pairs, where any *key* can have only one value associated at a time and there can not be equal keys. Moreover, the size of stored values are expected to be much larger than the size of the associated keys. We assume the presence of four operations: (1) **put**(k, v), (2) **get**(k), (3) **list**(), and (4) **remove**(k). The first operation associates a key k with the value v, returning *ack* if successful and *ERROR* otherwise; the second retrieves the value associated with a key k, or *ERROR* if the key does not exist; the third returns an array with all the keys in the collection, or [] if there are no keys in the collection; and the last operation disassociates a key k from its value, releasing storage space and the key itself, returning an *ack* if successful and *ERROR* otherwise. Finally, we assume that individual KVS's operations are atomic and wait-free.

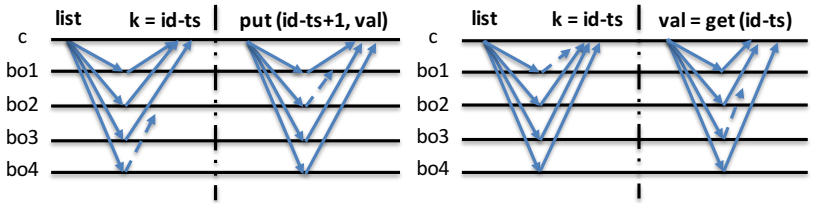
4 Multi-Writer Constructions

In this section we describe the three MW-regular register implementations. Before discussing the algorithms in detail (§4.3 to §4.6), we present an overview of the general structure of the protocols (§4.1) and describe the main techniques employed in their construction (§4.2). The correctness proofs of the protocols are presented in the extended version of this paper [29].

4.1 Overview

Our three MW-regular protocols differ mainly in the storage technique employed (replication or erasure code), the number of base objects required ($3f + 1$ or $4f + 1$), and the number of sequential base object accesses (two or three steps). Excluding these differences, the general structure of all protocols is similar to the one illustrated in Figure 1.

In the write operation, the client first lists a quorum of base objects (KVSs) in order to find the key encoding the most recent version written in the system, and then puts the value being written associated with a unique key encoding a new (incremented) version in a quorum. The read operation requires finding the most recent version of the object (as in the first phase of the write operation), and then retrieving the value associated with that key.



(a) WRITE **(b)** READ

Figure 1 General structure of our MW-regular register emulations.

Notice that our approach considers that each written value requires a new key-value pair in the KVSs. However, it is impossible to implement wait-free data-centric MW-regular register emulations without using at least one "data element" per written version if the base objects do not provide conditional update primitives (similar to Compare-and-Swap) [5, 11]. Therefore, any practical implementation of these algorithms must consider some form of garbage collection, as discussed in §5.2.

4.2 Protocols Mechanisms

Our algorithms use a set of mechanisms that are crucial for achieving Byzantine fault tolerance, MW semantics and storage efficiency. To simplify the exposition of the algorithms in the following sections (§4.4 to §4.6), we first describe such mechanisms.

4.2.1 Byzantine Quorum Systems

Our protocols employ dissemination and masking Byzantine quorum systems to tolerate up to f Byzantine faults [25]. *Dissemination quorum systems* consider quorums of $q = \lceil \frac{n+f+1}{2} \rceil$ base objects, requiring thus a total of $n > 3f$ base objects in the system. This ensures each two quorums intersect in at least $f + 1$ objects (one correct). *Masking quorum systems* require quorums of size $q = \lceil \frac{n+2f+1}{2} \rceil$ and a total of $n > 4f$ base objects, ensuring thus quorum intersections with at least $2f + 1$ base objects (a majority of correct ones).

4.2.2 Multi-Writer Semantics

We use the **list** operation of KVSs to design MW uniform implementations. This operation is very important as it allows us to discover new versions written by unknown clients. With this, the key idea of our protocols is making each writer to write in its own abstract register in a similar way to what is done in traditional transformation of SW to MW registers [23]. We achieve this by putting the client unique id on each key alongside with a timestamp ts, resulting in the pair $\langle ts, id \rangle$, which represents a *version*. This approach ensures that clients writing new versions of the data never overwrite versions of each other.

4.2.3 Object integrity and authenticity

We call the pair $\langle data\ key, data\ value \rangle$ an *object*. In our algorithms, the *data key*[2] is represented by a tuple $\langle ts, id, h \rangle_s$, where $\langle ts, id \rangle$ is the version, h is a cryptographic hash of the *data value* associated with this key, and s is a signature of $\langle ts, id, h \rangle$ (there is a slight difference in

[2] For the remaining of this paper we may refer to this only as *key*.

the protocol of §4.6, as will be discussed later). Having all this information on the data key allows us to validate the integrity and authenticity of the version (obtained through the **list** operation) before reading the data associated with it. Furthermore, if some version has a valid signature we call it *valid*. A data value is said to be *valid* if its hash matches the hash present in a valid key (this can only be verified after reading the value associated with the key). Consequently, an object is valid if both the version and the value are valid.

4.2.4 Erasure codes

Two of our protocols employ erasure codes [30] to decrease the storage overhead associated with full replication. This technique generates n different coded blocks, one for each base object, from which any $m < q$ base objects blocks can reconstruct the data. Concretely, in our protocols we use $m = f + 1$.

Notice that this formulation of coded storage can also be used to ensure confidentiality of the stored data, by combining the erasure code with a secret sharing scheme [22], in the same way it was done in DepSky [6].

4.3 Pseudo Code Notation and Auxiliary Functions

We use the '+' operator to represent the concatenation of strings and the '.' operator to access data key fields. We represent the parallelization of base object calls with the tag **concurrently**. Moreover, we assume the existence of a set of functions:

(1) *H(v)* generates the cryptographic hash of v;
(2) *encode(v, n, m)* encodes v into n blocks from which any m are sufficient to recover it;
(3) *decode(bks, n, m, h)* recovers a value v by decoding any subset of m out-of n blocks from the array bks if $H(v) = h$, returning $\perp$ otherwise;
(4) *sign(info, K_r)* signs *info* with the private key K_r, returning the resulting signature s;
(5) *verify(s, K_u)* verifies the authenticity of signature s using a public key K_u.

Besides these cryptographic and coding functions, our algorithms employ three auxiliary functions, described in Algorithm 1. The first function, *listQuorum* (Lines 1-6), is used to (concurrently) list the keys available in a quorum of KVSs. It returns an array L with the result of the **list** operation in at least q KVSs.

The *writeQuorum*(*data_key*, *value*) function (Lines 7-11) is used for clients to write data in a quorum of KVSs. The key *data_key* is equal in all base objects, but the value *value*[i] may be different in each base object, to accommodate erasure-coded storage. When at least q successful **put** operations are performed, the loop is interrupted.

The last function, *maxValidVersion*(L) finds the maximum version number correctly signed on an array L containing up to n KVS' **list** results (possibly returned from *listQuorum* function), returning 0 (zero) if no valid version is found.

4.4 Two-Step Full Replication Construction

Our first Byzantine fault-tolerant MW-regular register construction employs full replication, storing thus the entire value written in each base object. The algorithm is optimally resilient as it employs a dissemination quorum system [25]. Algorithm 2 presents the write and read procedures for the construction.

Processes perform write operations using the procedure **FR-write** (Lines 1–7). The protocol starts by listing a quorum of base objects (Line 2). Then, it finds the maximum version available with a valid signature in the result using the function *maxValidVersion*(L)

Algorithm 1: Auxiliary functions.

```
1  Function listQuorum() begin
2      L[0..n − 1] ←⊥;
3      concurrently for 0 ≤ i ≤ n − 1 do
4          L[i] ← list_i;
5      wait until |{i : L[i] ≠⊥}| ≥ q;
6      return L;
7  Function writeQuorum(data_key, value) begin
8      ACK[0..n − 1] ←⊥;
9      concurrently for 0 ≤ i ≤ n − 1 do
10         ACK[i] ← put(data_key, value[i])_i;
11     wait until |{i : ACK[i] = true}| ≥ q;
12 Function maxValidVersion(L) begin
13     return ⟨vr, h⟩_s ∈ ⋃_{i=0}^{n−1} L[i] : verify(s, K_u) ∧ ∄ ⟨vr′, h′⟩_{s′} ∈ ⋃_{i=0}^{n−1} L[i] : vr′ > vr ∧ verify(s′, K_u)) ;
```

(Line 3), and creates the new data key by concatenating a new unique version, and the hash of the value to be written together with the signature of these fields (Lines 4–5). Lastly, it uses the *writeQuorum* function to write the data to the base objects (Lines 7).

The read operation is represented in the **FR-read** procedure (Lines 8–22). As in the write operation, it starts by listing a quorum of base objects. Then the reader enters in a loop until it reads a valid value (Line 10–21). First, it gets the maximum valid version listed (Line 11), and then it triggers n parallel threads to read that version from different KVSs. Next, it waits either for a valid value, which is immediately returned, or for a quorum of q responses (Line 19). The only way the loop terminates due to the second condition is if it is trying to read a version being written concurrently with the current operation, i.e., a version that is not yet available in a quorum. This is possible if the first q base objects to respond do not have the maximum version available yet. When this happens, the version is removed from the result of the **list** operation (Line 20), and another iteration of the outer loop is executed to fetch a smaller version. Notice that a version that belongs to a complete write can always be retrieved from the inner loop due to the existence of at least one correct base object in the intersection between Byzantine quorums.

Without concurrency, the protocol requires one round of **list** and one of **put** for writing, and one round of **list** and one of **get** for reading. In fact, it is impossible to implement a MW register with fewer object calls since for writing and reading we always need to use at least one round of **put** and **get** operations, respectively, and to find the maximum version available we can only use **list** or **get** to retrieve that information from the base objects.

4.5 Two-Step Erasure Code Construction

Differently from the protocol described in the previous section, which employs full replication with a storage requirement of $q \times S$ wherein S is the size of the object, in our second Byzantine fault-tolerant MW-regular register emulation we use storage-optimal erasure codes. Since the erasure code we use [30] generates n coded blocks, each with $\frac{1}{f+1}$ of the size of the data, the storage requirement is reduced to $q \times \frac{S}{f+1}$.

The main consequence of storing different blocks in different base objects for the same version, is that the number of base objects accessed in dissemination quorum systems is not enough to construct a wait-free Byzantine fault-tolerant MW-regular register. This happens because the intersection between dissemination quorums contains only $f + 1$ base objects,

Algorithm 2: Regular Byzantine Full Replication (FR) MW register ($n > 3f$) for client c.

```
1  Procedure FR-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← value;
7      writeQuorum(data_key, v);
8  Procedure FR-read() begin
9      L ← listQuorum();
10     repeat
11         data_key ← maxValidVersion(L);
12         d[0..n − 1] ←⊥;
13         concurrently for 0 ≤ i ≤ n − 1 do
14             value_i ← get(data_key)_i;
15             if H(value_i) = data_key.hash then
16                 d[i] ← value_i;
17             else
18                 d[i] ← ERROR;
19         wait until (∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR) ∨ (|{i : d[i] ≠⊥}| ≥ q);
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until ∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR;
22     return d[i];
```

meaning that when reading the version associated with the last complete write operation, the quorum accessed may contain only 1 valid response (f can be faulty). This is fine for full replication as a single updated and correct value is enough to complete a read operation. However, it may lead to a violation of the regular semantics when erasure codes are employed since we now need at least $f + 1$ encoded blocks to reconstruct the last written value.

To overcome this issue, we use Byzantine masking quorum systems [25], where the quorums intersect in at least $2f + 1$ base objects. Despite the increase in the number of base objects ($n > 4f$), the storage requirement is still significantly reduced when compared with the previous protocol. As an example, for $f = 1$, this protocol has a storage overhead of 100% (a quorum of four objects with coded blocks of half of the original data size) while in the previous protocol the overhead is 200% (a quorum of three objects with a full copy of the data on each of them).

Algorithm 3 presents this protocol. The **EC-write** procedure is similar to the write procedure of Algorithm 2. The only difference is the use of erasure codes to store the data. Instead of full replicating the data, it uses the *writeQuorum* function to spread the generated erasure-coded blocks through the base objects in such a way that each one of them will store a different block (Lines 6–7). Notice that the hash on the data key is generated over the full copy of data and not over each of the coded blocks.

The read procedure **EC-read** is also similar to the read protocol described in §4.4, but with two important differences. First, we remove from L the versions we consider impossible to read (Lines 10–11), i.e., versions that appear in less than $f + 1$ responses. Second, instead of waiting for one valid response in the inner loop, we wait until we can reconstruct the data or for a quorum of responses. Again, the only way the loop terminates through the second condition is if we are trying to read a concurrent version. For reconstructing the original data, every time a new response arrives we try to decode the blocks and verify the integrity of the obtained data (Line 18). Notice that the integrity is verified inside the *decode* function. A version associated with a complete write can always be successfully decoded because any

Algorithm 3: Regular Byzantine Erasure-Coded (EC) MW register ($n > 4f$) for client c.

```
1  Procedure EC-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← encode(value, n, f + 1);
7      writeQuorum(data_key, v);
8  Procedure EC-read() begin
9      L ← listQuorum();
10     foreach ver ∈ L : #_L(ver) < f + 1 do
11         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {ver};
12     repeat
13         data_key ← maxValidVersion(L);
14         data ←⊥;
15         concurrently for 0 ≤ i ≤ n − 1 do
16             d[i] ← get(data_key)_i;
17             if data =⊥ then
18                 data ← decode(d, n, f + 1, data_key.hash);
19         wait until data ≠⊥ ∨ |{i : d[i] ≠⊥}| ≥ q;
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until data ≠⊥ ∧ data ≠ ERROR;
22     return data;
```

accessed quorum will provide at least $f + 1$ valid blocks for decoding this version's value. As soon as the integrity is verified, the outer loop stops and the value is returned (Lines 21–22).

4.6 Three-Step Erasure Code Construction

Our last construction implements a Byzantine-resilient MW-regular register using erasure codes and dissemination quorums, being thus both storage-efficient and optimally-resilient. We achieve this by storing in each base object two objects per version instead of one. The first one, the *data object*, is used to store the encoded data blocks. The second one, the *proof object*, is an object with a zero-byte value used to prove that a given data object is already available in a quorum of base objects (similar to what is done in previous works [14, 6]). The key of the data object is composed only by the version, i.e., the tuple $\langle ts, id \rangle$. In turn, the key of the proof object is composed by the string $\langle$"PoW", $ts, id, h\rangle_s$, in which h is the hash of the full copy of data and s is a signature of $\langle$"PoW", $ts, id, h\rangle$.

Algorithm 4 presents the protocol. The write procedure, called **3S-write**, starts by listing the proof objects from a quorum of base objects (Line 2). Then, it finds the maximum valid version between the proof objects. For simplicity, this algorithm uses the same function $maxValidVersion(L)$ as the previous protocols, but here we are only interested in proof objects. Next, it creates the new data key and the new proof key to be written (Lines 4–6). Then it writes the data object in a quorum (ensuring that different base objects will store different coded blocks) and, after that, it writes the proof object (Lines 7-10). This sequence of actions ensures that when a valid proof object is found in at least one base object, the corresponding data object is already available in a quorum of base objects.

The **3S-read** procedure is used for reading. The idea is to list proof objects from a quorum, find the maximum valid version among them, and read the data object associated with that proof object. Notice that to read the data we do not need to wait for a quorum of responses as it is enough to have $m = f + 1$ valid blocks to decode the value (Lines 18–19). This holds because, differently from the two previous algorithms, here we are sure that the

Algorithm 4: Regular Byzantine Erasure-Coded (EC) MW register ($n > 4f$) for client c.

```
1  Procedure 3S-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      data_key ← ⟨max.ts + 1, c⟩;
5      proof_info ←"PoW"+⟨max.ts + 1, c, H(value)⟩;
6      proof_key ← proof_info + sign(proof_info, K_r) ;
7      v[0..n − 1] ← encode(value, n, f + 1);
8      writeQuorum(data_key, v);
9      v[0..n − 1] ← ∅;
10     writeQuorum(proof_key, v);
11 Procedure 3S-read() begin
12     L ← listQuorum();
13     proof_key ← maxValidVersion(L);
14     data_key ← ⟨proof_key.ts, proof_key.id⟩;
15     data ←⊥;
16     concurrently for 0 ≤ i ≤ n − 1 do
17         d[i] ← get(data_key)_i;
18         if data =⊥ then
19             data ← decode(d, n, f + 1, data_key.hash);
20     wait until data ≠⊥;
21     return data;
```

data values with a version matching the maximum version found in valid proof objects is already stored in a quorum of base objects.

As explained before, this protocol works with only $3f + 1$ base objects. This is done without adding any extra call to the base objects in the read operation, which still needs only two rounds of accesses, one for **list** and one for **get**. However, for writing, one additional round of **put** is needed (to replicate the proof object). This trade-off is actually profitable in a cloud-of-clouds environment since the monetary costs of storing erasure-coded blocks in extra clouds is much larger than sending zero-byte objects to the clouds we use.

5 Protocols Extensions

This section presents a discussion of how the protocols presented in this paper can be modified to offer atomic semantics [23], and what are the possible solutions to garbage collect obsolete data versions.

5.1 Atomicity

There are many known techniques to transform regular registers in atomic ones. Most of them require servers running part of the protocol [9, 27], which is impossible to implement with our base objects. Fortunately, the simplest transformation can be used in data-centric algorithms. This technique consists in forcing readers to *write-back* the data they read to ensure this data will be available in a quorum when the read completes [17, 26, 5].

Our three read constructions could implement this technique by invoking *writeQuorum* to write the read value before returning it. However, writing back read values in our first two protocols may carry out performance issues as the stored data size might be non-negligible. In turn, employing the same write-back technique in our last protocol (Algorithm 4) does not have such overhead, as a reader would only need to write-back the small proof object (see §4.6). Hence, the performance effect of using this technique in the read procedure is independent of the size of the data being read.

A final concern about using write-backs to achieve atomicity is that we would have to assume that readers may only fail by crash, otherwise they may write bogus values in the base objects. In the regular constructions this is not required as we do not need to give write permissions to readers.

5.2 Garbage Collection

Existing solutions

Register emulations that employ versioning must use a garbage collection protocol to remove obsolete versions, otherwise an unbounded amount of storage is required. DepSky [6] provides a garbage collection protocol that is triggered periodically to remove older versions from the system. Although practical in many applications (e.g., cloud-backed file system [7]), this solution is vulnerable to the *garbage collection racing problem* [5, 33]. This problem happens when a client is reading a version that had became obsolete due to a concurrent write, and removed by a concurrent execution of the garbage collection protocol, making it impossible for a reader to obtain the value associated with it.

To the best of our knowledge, there are only two works that solve this problem. The solution of [33] makes readers announce the version they are going to read, preventing the garbage collector from deleting it. Unfortunately, this solution cannot be directly applied in the data-centric model since it requires servers capable of running parts of the algorithm. Another solution was proposed in [5]. In this protocol each writer stores the value in a *temporary* key, which can be garbage collected by other writers, and also in an *eternal* key, that is never deleted. This approach allows readers to obtain the value from the eternal key when the temporary key is erased by concurrent writers. A solution like this can be applied to our first protocol (see §4.4), which employs full replication. Yet, it does not work with erasure-coded data. This happens because the eternal key is overwritten whenever a write operation occurs, and since several writers can operate simultaneously, the eternal key in different base objects may end up with blocks belonging to different versions. Therefore, it might lead to the impossibility of getting $f + 1$ blocks of the same version to reconstruct the original value.

Adapting the solutions to our protocols

All existing solutions for garbage collection can be adapted to the protocols discussed in §4. The approach of deleting obsolete versions asynchronously by a thread running in background can be naturally integrated to our protocols. This thread can be triggered by the clients at the end of the write operations, making each client responsible for removing its obsolete data.

Since we do not rely on server-side code for our protocols, devising a solution where readers announce the version they are about to read (by writing an object with that information to a quorum of base objects) would require substantial changes in our system model. More specifically, to ensure wait-freedom for read operations, only objects with versions lower than the ones announced can be garbage collected. This solution may not tolerate the crash of the readers – if a reader crashes without removing its announcement, larger versions than the one it announced will never be removed. It is possible to add an expiration time to the announcement to avoid this. Yet, this would still require changes in the system model to add synchrony assumptions for the expiration time to (eventually) hold, and not consider Byzantine readers (that could block garbage collection by announcing the intention to read all versions).

Using the eternal key approach together with erasure codes significantly increases the storage requirements of our algorithms. The idea is to make each writer not only to store the coded blocks into temporary keys, but also to replicate full copies of the original data in eternal keys. This approach may lead to a decrease in the write performance (related with an extra write of a full copy of the data per base object) and an increase of $n \times S$ in each protocol storage requirements.

Discussion

The three proposed solutions explore different points in the design space of data-centric storage protocols. In the first approach, we do not really solve the garbage collection racing problem. The second solution requires a stronger system model and additional base object accesses in the read operation. The third solution increases the storage requirements and reduces the write performance as writers have to write not only the coded blocks, but also full copies of the data.

We argue that most applications would prefer to have better performance and a reduced storage complexity, at the cost of eventually repeating failed reads. Therefore, we chose to support the asynchronous garbage collection triggered periodically (for example hourly, daily or even when a given number of versions has been written), as done in DepSky [6].

6 Evaluation

This section presents an evaluation of our three new protocols, comparing them with two previous constructions targeting the cloud-of-clouds model [5, 6].

6.1 Setup and Methodology

The evaluation was done using a machine in Lisbon and a set of real cloud services. This machine is a Dell Power Edge R410 equipped with two Intel Xeon E5520 (quad-core, HT, 2.27Ghz), and 32GB of RAM. This machine was running an Ubuntu Server Precise Pangolin operative system (12.04 LTS, 64-bits, kernel 3.5.0-23-generic), and Java 1.8.0_67 (64-bits).

Furthermore, we compare our protocols with the MW-regular register of [5], which we call ICS, and the SW-regular register of DepSky (the DepSky-CA algorithm) [6]. The protocols proposed in this paper were implemented in Java using the APIs provided by real storage clouds. We used the DepSky implementation available online [13]. However, since there is no available implementation of ICS, we implemented it using the same framework we used for our protocols. All the code used in our experiments is available on the web [28].

All experiments consider $f = 1$ and the presented results are an average of 1000 executions of the same operation, employing garbage collection after every 100 measurements. The storage clouds used were Amazon S3 [3], Google Storage [18], Microsoft Azure Storage [10], Rackspace Cloud Files [31], and Softlayer Cloud Storage [32]. ICS was configured to use the first three of them ($n = 3$); the Two-Step Full Replication (2S-FR), Three-Step Erasure Codes (3S-EC) and DepSky protocols used the first four clouds mentioned ($n = 4$); and the Two-Step Erasure Codes (2S-EC) protocol used all of them ($n = 5$).

6.2 List Quorum Performance

One of the main differences between our protocols and the other MW-regular register of the literature designed for KVSs, namely ICS [5], is that in our algorithms the garbage collection is decoupled from the write operations. Since in ICS the garbage collection is included in the

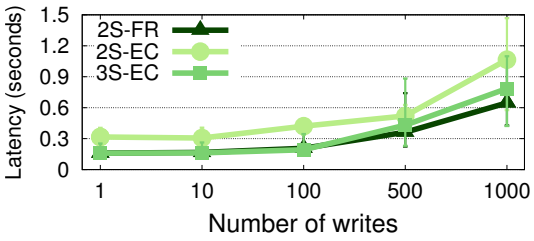
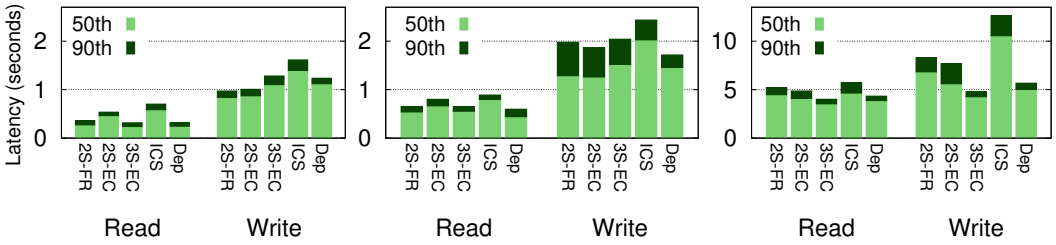


Figure 2 Average latency and std. deviation of *listQuorum* for different number of stored keys.



(a) 64kB **(b)** 1MB **(c)** 16MB

Figure 3 Median and 90-percentile latencies for read and write operations of register emulations.

write procedure, the **list** operation invoked in its base objects always return a small number of keys. However, as in our protocols the garbage collection is executed in background, it is important to understand how the presence of obsolete keys (not garbage collected) in the KVSs affects the latency of listing the available keys. Notice this issue does not affect DepSky as it does not use the **list** operation [6].

Figure 2 shows the latency of executing the *listQuorum* function with different numbers of keys stored in the KVSs, for our three protocols (which consider different quorum sizes). As can be seen, 2S-EC presents the worst performance, indicating that listing bigger quorums is more costly. We can also observe that the performance degradation of the **list** operation when there are less than 100 obsolete versions is very small (specially for 2S-FR and 3S-EC). However, the latency is roughly 2× and 4× worse when listing 500 and 1000 versions, respectively. This suggests that triggering the garbage collection once every 100 write operations will avoid any significant performance degradation.

6.3 Read and Write Latency

Figure 3 shows the write and read latency of our protocols, ICS [5] and DepSky [6], considering different sizes of the stored data.

The results show that, when reading 64kB and 1MB, 2S-FR and 3S-EC present almost the same performance, while 2S-EC is slightly slower, due to the use of larger quorums. This means that reading only one data value with a full copy of the data is as fast as reading $f+1$ blocks with half of the size of the original data. This is not the case for 16MB data. The results show it is faster to read $f+1$ data blocks of 8MB in parallel from different clouds (2S-EC and 3S-EC) than reading a 16MB object from one cloud (2S-FR).

For writing 64kB objects 3S-EC is slower than 2S-FR and 2S-EC. This happens due to the latency of the third step of the protocol (write of the proof object). When writing 1MB objects, our protocols present roughly the same latency, being the 3S-EC protocol a little bit slower (also due to the write of the proof object). However, when clients write 16MB

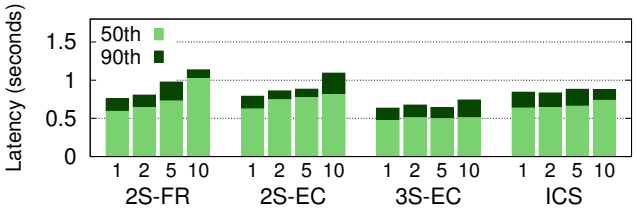


Figure 4 Median and 90-percentile read latencies in presence of contending writers.

data objects, the additional latency associated with this third step is negligible. Overall, these results can be explained by the fact that the proof object has zero bytes. Thus, 3S-EC protocol presents the best performance due to its use of dissemination quorums and erasure codes. For this data size, the 2S-FR protocol presents the worst performance of our protocols as it stores a full copy of the data in all clouds.

The key takeaway here is that our protocols present a performance comparable with DepSky [6] (Dep), *which does not support multiple writers*, and a performance up to 2× better than the *crash fault-tolerant* MW register presented in [5] (ICS). On the other hand, ICS presents the worst latency among the evaluated protocols. One of the main reasons for this to happen is because it does not use erasure codes. Furthermore, for reading, this protocol always waits for a majority of data responses, which makes it slower than, for example, the 2S-FR that only waits for one valid **get** response. In turn, for writing, ICS writes the full copy of the data twice on each KVS to deal with the garbage collection racing problem, removing also obsolete versions.

6.4 Read Under Write Contention

Figure 4 depicts the read latency of 1 MB objects in presence of multiple contending writers. This experiment does not consider DepSky as it only offers SW semantics.

The results show that both 2S-FR and 2S-EC read latencies are affected by the number of contending writers. This happens for two reasons: (1) under concurrent writes, these protocols typically try to read incomplete versions from the KVSs before finding a complete one (i.e., the loop on read protocols is executed more than once); (2) since we are not garbage collecting obsolete versions, more writers send more versions to the clouds, negatively influencing the *listQuorum* function latency. Since 3S-EC is not affected by the first factor, its read operation performs slightly better with contending writers.

ICS's read presents a constant performance with the increase of contending writers, however, 2S-FR and 2S-EC present competitive results and 3S-EC presents results always better than it, even without garbage collecting obsolete versions.

7 Conclusion

This paper considers the study of fundamental storage abstractions resilient to Byzantine faults in the data-centric model, with applications to cloud-of-clouds storage. In this context, we presented three new register emulations: (1) one that uses dissemination quorums and replicates full copies of the data across the clouds, (2) another that uses masking quorums and reduces the space complexity through the use of erasure codes, and (3) a third one that increases the number of accesses made to the clouds to use dissemination quorums together with erasure codes.

Our evaluation shows that the new protocols have similar or better performance and storage requirements than existing emulations that either support a single writer [5] or tolerate only crashes [6].

References

1 I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5), 2006.
2 M. Aguilera, B. Englert, and E. Gafni. On using network attached disks as shared memory. In *Proc. of the PODC*, 2003.
3 Amazon S3. URL: `http://aws.amazon.com/s3/`.
4 H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1), 1995.
5 C. Basescu et al. Robust data sharing with key-value stores. In *Proc. of the DSN*, 2012.
6 A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Transactions on Storage*, 9(4), 2013.
7 A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: a shared cloud-backed file system. In *Proc. of the USENIX ATC*, 2014.
8 C. Cachin, B. Junker, and A. Sorniotti. On limitations of using cloud storage for data replication. In *Proc. of the WRAITS*, 2012.
9 C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. of the DSN*, 2006.
10 B. Calder et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proc. of the SOSP*, 2011.
11 G. Chockler, D. Dobre, A. Shraer, and A. Spiegelman. Space bounds for reliable multi-writer data store: Inherent cost of read/write primitives. In *Proc. of the PODC*, 2016.
12 G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1), 2005.
13 DepSky webpage. URL: `http://cloud-of-clouds.github.io/depsky/`.
14 D. Dobre, G. O. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolic. Powerstore: Proofs of writing for efficient and robust storage. In *Proc. of the CCS*, 2013.
15 E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1), 2003.
16 G. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *Proc. of the ASPLOS*, 1998.
17 G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proc. of the DSN*, 2004.
18 Google storage. URL: `https://developers.google.com/storage/`.
19 J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. of the SOSP*, 2007.
20 M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.
21 P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3), 1998.
22 Hugo Krawczyk. Secret sharing made short. In *Proc. of the CRYPTO*, 1993.
23 L. Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1), 1986.
24 R. Los, D. Shacklenford, and B. Sullivan. The notorious nine: Cloud Computing Top Threats in 2013. Technical report, Cloud Security Alliance (CSA), February 2013.
25 D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.
26 D. Malkhi and M. K. Reiter. Secure and scalable replication in Phalanx. In *Proc. of the SRDS*, 1998.

27 J.P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. of the DISC*, 2002.

28 MWMR-registers webpage. URL: `https://github.com/cloud-of-clouds/mwmr-registers/`.

29 T. Oliveira, R. Mendes, and A. Bessani. Exploring key-value stores in multi-writer Byzantine-resilient register emulations. Technical Report DI-FCUL-2016-02, ULisboa, 2016.

30 M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), 1989.

31 Rackspace cloud files. URL: `http://www.rackspace.co.uk/cloud/files`.

32 Softlayer Cloud Storage. URL: `http://www.softlayer.com/Cloud-storage/`.

33 Y. Ye, L. Xiao, I-L. Yen, and F. Bastani. Secure, dependable, and high performance cloud storage. In *Proc. of the SRDS*, 2010.

The Case for Reconfiguration without Consensus: Comparing Algorithms for Atomic Storage

Leander Jehl[1] and Hein Meling[2]

1 **University of Stavanger, Stavanger, Norway**
`leander.jehl@uis.no`
2 **University of Stavanger, Stavanger, Norway**
`hein.meling@uis.no`

Abstract

We compare different algorithms for reconfigurable atomic storage in the data-centric model. We present the first experimental evaluation of two recently proposed algorithms for reconfiguration without consensus and compare them to established algorithms for reconfiguration both with and without consensus.

Our evaluation reveals that the new algorithms offer a significant improvement in terms of latency and overhead for reconfiguration without consensus. Our evaluation also shows that reconfiguration without consensus, can obtain similar results to that of consensus-based reconfiguration, which relies on a stable leader. Moreover, the new algorithms also substantially reduces the overhead compared to consensus-based reconfiguration without a leader.

While our analysis confirms our intuition that batching reconfiguration requests serves to reduce the overhead of reconfigurations, our evaluation also shows that it is equally important to separate reconfigurations from read and write operations. Specifically, we found that using read and write operations to assist in completing concurrent reconfigurations is in fact detrimental to the reconfiguration performance.

1998 ACM Subject Classification C.3.4 Distributed Systems, C.4 Performance of Systems

Keywords and phrases atomic storage, reconfiguration, data-centric model

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.31

1 Introduction

Cloud computing facilities offers an abundance of compute resources located in data centers around the globe. These data centers can host a wide range of services with different requirements and characteristics. For example, some services require replication for fault-tolerance. However, managing these data centers can be challenging, and often administrators will need to update both the composition of machines in the data center and the composition of replicas running a service. This is necessary to replace failed components, upgrade machine hardware, and adapt to changes in the service load. In practice, such reconfiguration operations are relatively frequent, as is evident from the traces of a Google data center [18].

To support reconfiguration, one of the main challenges faced by data center operators is to ensure consistency when multiple users submit concurrent reconfiguration requests. Moreover, a multitude of components may be monitoring software and hardware failures, upgrades, and the load of queries and updates to the replicas. Acting upon this information, reconfiguration requests may be issued autonomously, without human intervention [2]. We envisioned that many such components may be deployed in a large-scale data center at the

same time, which may result in multiple concurrent uncoordinated and even conflicting requests for reconfiguration.

Rambo [12] was the first system to address reconfiguration of atomic storage. They used consensus to decide on reconfigurations. In [1] it was shown that such reconfiguration is possible in an asynchronous system, without having to solve consensus. However, experimental results [20] have shown that such reconfigurations add a significant overhead to concurrent read and write operations compared to reconfigurations using consensus. Recently however, two new approaches for reconfiguration without consensus have been proposed [14, 11]. These rely on lattice agreement (LA) [8] or a speculating snapshot algorithm (SpSn) [11] to reduce the worst case communication complexity of reconfigurations and operations concurrent with reconfigurations. While these two approaches appear to be superior in theory, understanding their behavior in different deployment scenarios is not obvious, since no experimental evaluation has been done.

Our main contribution in this paper is an extensive experimental evaluation of several algorithm variants of reconfigurable storage, SmartMerge-Store (SM-Store) [14], SpSn-Store [11], DynaStore [1], and Rambo [12]. SM-Store and SpSn-Store implement the novel approaches using lattice agreement and speculating snapshot, respectively.

Further, these reconfiguration algorithms are often presented in different models and formalisms. This has made it difficult to comprehend and compare the different algorithms.

Hence, our second contribution in this paper is a presentation of these algorithms, based on a common template aimed at highlighting their most relevant differences and similarities.

We implemented the algorithms in the data-centric model, in which processes are strictly separated into client and server roles, prohibiting servers from initiating communication. Section 2 describes this model in detail and motivate our choice.

In a local area network, our evaluation shows that, compared to DynaStore, the new algorithms' ability to batch reconfiguration requests can significantly reduce the overhead imposed on concurrent read operations. The new algorithms also have lower overhead than Rambo, if run without a stable leader, and similar overhead to leader-based Rambo.

Our evaluation indicates that treating read and write operations separately from reconfigurations is an important design principle. Different from SM-Store and Rambo, the SpSn-Store and DynaStore algorithms force read and write operations to help [4] in completing concurrent reconfigurations. However, we found that this actually increases the overhead that reconfigurations impose on read and write operations.

We implemented an optimization from Rambo, that allows a client to reuse a server's reply in the context of different configurations. We call this *single contact mode*. Our results show that this optimization can significantly reduce the overhead for read operations.

We also evaluated the algorithms for the case where clients and servers reside in different data centers across the globe. In this scenario, we found that the impact of batching reconfigurations is less pronounced, and in some cases DynaStore actually performs better than the new algorithms.

2 Why the Data-Centric Model?

The different algorithms in our study have been proposed and studied in different models. SpSn-Store [11] was presented in the data-centric model, while Rambo [12], SM-Store [14] and DynaStore [1] were all presented in the process-centric model. However, to the best of our knowledge, the only evaluation of reconfigurable storage without consensus was done in the data-centric model [20]. This section presents the two models, and motivate our choice of the data-centric model.

In the process-centric model, the processes invoking operations, e.g. reading and writing, also maintain the stored state and can respond to requests from other processes.

In the data-centric model, we distinguish between client and server processes. Clients perform operations, while servers respond to client requests and maintain state. Clients and servers can, but need not be located on the same machines. This model restricts communication, in that servers only respond to client requests. That is, servers cannot initiate communication with clients and two servers do not communicate directly.

The data-centric model is generally considered to be more scalable [20, 6] since it reduces bottlenecks and avoids all-to-all message patterns common in the process-centric model. Moreover, the set of clients can easily be changed without changing the set of servers.

It is not clear how to compare latencies and throughput achieved in the different models. Furthermore, in DynaStore and SM-Store if a process is removed while performing an operation, this operation may never complete. This makes it difficult to measure the latencies of operations concurrent with reconfigurations. We therefore implemented all algorithms in the data-centric model. The algorithms in our study are all a good match for this model, since their interactions mostly follow the request-reply pattern between clients and servers. With this choice, our results are also directly comparable with the results from [20]. This is relevant, since part of our motivation is to test whether the new protocols for reconfiguration without consensus (SpSn-Store and SM-Store), also introduce the significant overhead to *read* and *write* operations, as observed in [20].

However, in the data-centric model an idle client cannot be informed by the servers, when one or more reconfigurations together replace all of the servers with new ones. This problem arises in many reconfiguration algorithms and is typically solved using a resource discovery service [20, 17, 21]. Even in systems where idle clients are notified, a resource discovery service is still needed to allow new clients to join. Since our evaluation is focused on the *read* and *write* performance during reconfiguration, we refer to these other works for solutions to the discovery problem.

To use a **single leader** to propose reconfigurations is an easy way to avoid concurrent reconfigurations, and thus the main difficulty of reconfiguration. This is especially relevant for Rambo, which uses consensus to choose one of several concurrent reconfigurations, because an established leader can skip the first phase of consensus [16].

To ensure a fair comparison for Rambo's consensus-based algorithm, we have implemented two variants of this algorithm; one where clients forward reconfiguration requests to a leader, and one where every process believes itself to be the leader. We refer to the leader variant as L-Rambo. Note that L-Rambo does not entirely comply with the data-centric model. A leader must both receive requests from other clients, and perform operations on the servers. It is therefore both a client and a server. Accordingly, we found that introducing the additional role of a leader significantly increased the complexity of implementing and deploying this variant. While this assessment is clearly subjective, for us it validated the claim, that the data-centric model promotes simplicity. The other algorithms, besides Rambo, could also benefit from a leader batching reconfigurations. However, we believe it is more interesting to compare the leaderless algorithms with L-Rambo.

3 Reconfigurable Storage Interface

The algorithms in our study provide three operations, *read*, *write*, and *reconf*. The *write* operation stores a single value and the *read* operation returns the last value that was written. The *reconf* operation is used to change the set of servers and is discussed below. In all algorithms in our study the *read* and *write* operations fulfill atomic [15] semantics. Thus,

even if several operations are executed concurrently, they appear as if all operations where executed sequentially.

The first algorithm to implement atomic *read* and *write* objects in an asynchronous system, that is subject to failures, was the ABD algorithm [3]. All algorithms in our study are based on this work. In the ABD algorithm, values are always stored together with a logical timestamp. We refer to such a (timestamp, value) pair as a timestamped value, and say that one timestamped value is greater than another, if it has a higher timestamp. The ABD algorithm assumes a fixed set of servers. Values are read from and written to a majority of these servers.

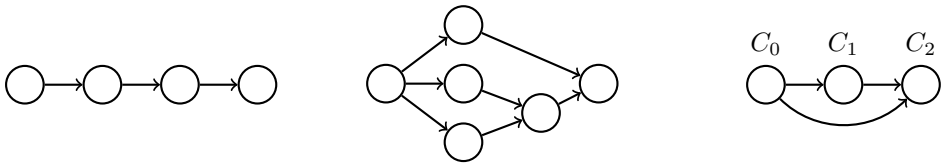
In ABD, both *read* and *write* operations proceed in similar manner, performing first a query, followed by a propagation phase. During the query phase a client collects timestamped values from a majority of the servers. Among the collected values, the client determines the one with the highest timestamp. In a *read* operation, before returning this value, the client propagates the value and timestamp back to a majority, to ensure that successive operations will also read this value. A client performing a *write* operation, on the other hand, uses the highest timestamp found in the query phase along with its process identifier, to create a higher, unique timestamp, and propagates its own value, together with the new timestamp to a majority of servers.

Thus, *read* and *write* operations only differ in the value and timestamp they propagate, not in the actual set of messages that need to be sent and received. This is also true for the reconfigurable algorithms in this study. The ABD algorithm can be optimized to allow some reads to return after the query phase [7]. The applicability of such optimizations depends on the workload, since usually only reads that are not concurrent with a *write* operation can be optimized. Similarly, a regular *read* operation only performs the query phase of the atomic *read* operation, but provides weaker consistency [19]. In our study, we have implemented regular reads, to enable comparison with the most impactful optimization.

There also exist optimizations for managing large objects, e.g. by separating stored objects from metadata [9]. Using the algorithm from [9], the reconfiguration of servers storing the actual data is easy and its performance mainly depends on the size of the state. The algorithms in this study could be used for reconfiguration of metadata servers.

All algorithms organize the servers into configurations. A configuration is a set of servers. In SM-Store and Rambo, a configuration also includes a read-write quorum system, while DynaStore and SpSn-Store assumes that a majority quorum is used. In our implementations of SM-Store and Rambo a write-quorum consists of a majority of the processes in a configuration, while a read-quorum consists of at least half the processes in the configuration. This way, the quorum systems used in SM-Store and Rambo provide the same fault tolerance as the majority quorums used in DynaStore and SpSn-Store, while SM-Store and Rambo may still benefit from their ability to use a more flexible quorum system.

Besides *read* and *write*, all algorithms in our study allow a *reconf* operation to change the configuration. In DynaStore and SpSn-Store, this operation takes a set of tuples $(s_i, +)$ or $(s_j, -)$ as input, where $(s_i, +)$ signals that server s_i should be added, while $(s_j, -)$ removes s_j. We call a set $\{(s_i, +), (s_j, -), ...\}$ a *change*. DynaStore and SpSn-Store assume an initial configuration C_0. Every other configuration C_l is identified by a change ch_l, such that C_l results from applying ch_l to C_0. We write $C_l = ch_l(C_0)$. To apply an additional change ch to C_l we simply add ch to ch_l: $ch(C_l) = ch \cup ch_l(C_0)$. In these algorithms, after a *reconf*$(\{(s_1, +), (s_2, -)\})$ operation returns, server s_2 is no longer part of the current configuration, and s_1 is part of the configuration, unless it was explicitly removed by another reconfiguration.



(a) Rambo uses consensus to choose a successor for every configuration.

(b) In **DynaStore**, configurations can have multiple successors.

(c) Using **LA** or **SpSn**, we can ensure that configurations are ordered.

Figure 1 Directed acyclic graphs of successor configurations. Circles are configurations and arrows are established successors.

In the same way, SM-Store's *reconf* operation takes a change as argument. Furthermore, the *reconf* operation takes a policy that can be used to specify additional changes, e.g. to the quorum system. However in the context of this paper, we are interested in comparing the performance of reconfigurations that can be specified in all algorithms, and thus we always use an empty policy.

Rambo's reconfiguration interface differs from that of the other algorithms, where a reconfiguration only returns after its change has been applied. That is, a *rambo-reconf* proposes a new configuration C to a consensus instance in the current configuration *cur*. Only if C is chosen by this consensus, C will become the new current configuration. We therefore create a wrapper for *rambo-reconf* that has the same semantics as the other *reconf* operations. This *reconf* operation receives a change as argument, then applies this change to the current configuration *cur*, and invokes *rambo-reconf*(*change*(*cur*)). If some configuration C' was chosen by consensus that does not include the change proposed by our reconfiguration, we apply our change again, invoking *rambo-reconf*(*change*(C')). Thus a reconfiguring client may need to invoke several *rambo-reconf* operations before its change is applied and it can return from the *reconf* operation.

4 A Common Template for Reconfiguration Algorithms

We now present a common template for the different reconfiguration algorithms that we evaluate. We slightly simplified the algorithms for this presentation, to better highlight relevant similarities and differences. In our implementation we follow the original version of the algorithms.

4.1 The Graph of Successor Configurations

To apply a change *ch* to the current configuration C, a client must register this change with the servers in configuration C. Registered changes create a directed acyclic graph (DAG) of configurations, where an arc connects C to $ch(C)$, if the change *ch* was registered with C. In this case we say that $ch(C)$ is a successor of C. Figure 1 shows some examples.

Registering a change might fail. For example, in Rambo only a single change is chosen by a configuration (Figure 1a). Thus a reconfiguring client must traverse the graph, until it can record its changes. In the other algorithms a configuration may have multiple successors (Figures 1b and 1c). In this case, a client must traverse the graph to ensure that the new configuration is a direct or indirect successor of every other configuration in the graph.

Our common template for the reconfiguration algorithms, depicting such a graph traversal, is shown in Figure 2. In every visited configuration, the client tries to establish a new

successor configuration that includes the requested *change* and collects information on existing successors. The function ESTABLISH&COLLECTSUCCS(*change*) on Line 9 of the template is implemented differently by each algorithm, as we describe below. The client then updates the DAG with the collected successors on Line 12. The changes realized in these successors are also added to the proposed *change* on Line 13. This ensures that concurrent reconfigurations do not cancel each other, but eventually all reach the same configuration. The client collects the state from each visited configuration, and transfers the most up-to-date state to the new configuration (Lines 10–15). On Line 16 the client starts the new configuration. Future reconfigurations may then begin their traversal from this configuration and old configurations can be discarded.

We had to adjust how configurations are started, to fit with the data-centric model. The original versions of Rambo, SM-Store, and DynaStore use an all-to-all broadcast to inform all clients that a new configuration was started. This violates the assumptions of the data-centric model, which disallows broadcasting to all clients. In our implementation, the reconfiguring client performs *cur*.start() by informing the servers in *cur* that this configuration was started. When replying to other clients, the servers include this information, allowing also other clients to discard old configurations. This approach was also used in [20] to adapt start() to the data-centric model.

Algorithm 1 shows two common primitives that we use to describe the different algorithms. We assume that every server s_i in a configuration *cur* stores a set of proposed changes Ch_i. Each of these changes corresponds to a successor configuration *change*(*cur*). The *cur*.ADDCHANGES($\{ch_1, ch_2, ...\}$) primitive tells every server s_i in *cur* to add the changes ch_1 and ch_2 to its change set Ch_i, thus adding two new successor configurations. ADDCHANGES only returns after a majority of the servers have applied this update. Note that each of the changes ch_1 or ch_2 may include several additions and removals. However, only in DynaStore is ADDCHANGES actually called with a set of changes. Similarly, *cur*.GETCHANGES() reads the Ch_i variables at a majority of the processes in *cur* and returns the union of these sets.

For **Rambo**, the implementation of ESTABLISH&COLLECTSUCCS(*change*) is shown in Algorithm 2. The reconfiguring client proposes its *change* to the consensus instance in configuration *cur* and learns a possibly different set of changes *ch* that has been chosen by consensus. We use the Paxos consensus algorithm [16], however, we do not use all-to-all learn messages, since they do not comply with the data-centric model. Instead we only send learn messages to the client that proposed a value. After learning a new configuration from Paxos, the reconfiguring client then informs the servers in *cur*, that *ch* was chosen, invoking *cur*.ADDCHANGES(*ch*). It is not necessary to collect successors, since no other change than *ch* can be chosen by consensus.

We note that there exists an optimized variant of Rambo, called RDS [5]. In RDS the servers in an old configuration forward their state directly to all servers in the new configuration. This reduces the number of message delays necessary for a reconfiguration. We have not implemented this optimization, since it relies on an all-to-all message exchange among servers and is thus not applicable to the data-centric model.

Since Rambo only chooses a single successor for every configuration, the graph of configurations has a single path, as shown in Figure 1a. In L-Rambo, where a leader performs reconfigurations on behalf of other clients, this leader can combine the changes proposed by different clients in a single configuration and propose this to the consensus algorithm. We refer to this process as batching.

Without consensus it is not possible to choose a single successor. Thus in **DynaStore**, multiple successors can be established for one configuration. These successors must eventually

Template for reconfiguration.

```
1:  State :
2:      cur                                              ▷ current configuration
3:  reconf(change)
4:      allchanges := change                             ▷ set to collect all changes applied in this reconf
5:      DAG := cur                                       ▷ graph with single node
6:      S := {}                                          ▷ set of timestamped values
7:      while allchanges(cur) ≠ cur do
8:          cur ← next ∈ DAG in topological order
9:          Ch ← cur.ESTABLISH&COLLECTSUCCS(allchanges)  ▷ try to establish successor, collect successors
10:         S ← S ∪ cur.collectState()                   ▷ collect timestamped values from majority
11:         for ch ∈ Ch do
12:             DAG.add(cur → ch(cur))                   ▷ add successor arc to graph
13:             allchanges ← allchanges ∪ ch             ▷ combine changes
14:     v := max_{v.ts}(S)                               ▷ find value with highest timestamp
15:     cur.updateState(v)                               ▷ update timestamped value at majority
16:     cur.start()                                      ▷ Inform servers in cur that cur is started
```

Alg. 1 Auxiliary functions (RPCs to access state at servers)

```
State at every server s_i:
    Ch_i := {}                                   ▷ set of changes
function cur.ADDCHANGES({ch_1, ch_2, ...})
    for all s_i ∈ cur invoke in parallel
        at s_i do: Ch_i ← Ch_i ∪ {ch_1, ch_2, ...}   ▷ remote
    wait until assignment completed at majority in cur
function cur.GETCHANGES
    for all s_i ∈ cur invoke in parallel
        Ch_i := s_i.read(Ch_i)                   ▷ read from remote
    wait until s_i.read(Ch_i) completed at majority in cur
    return ⋃ Ch_i                                ▷ only those with completed read
```

Alg. 2 Rambo: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      ch ← cur.consensus.propose(change)   ▷ 1 to ∞ round trips
3:      cur.ADDCHANGES({ch})                 ▷ 1 round trip
4:      return {ch}
```

Alg. 3 SpSn-Store: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      return cur.SpSn(change)              ▷ 2 to 2r round trips
```

Alg. 4 SM-Store: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      ch_LA ← cur.LA(change)        } 1 to r
3:      Ch ← cur.GETCHANGES()         }
4:      if ∃ch ∈ Ch then              } 1 round trip
5:          cur.ADDCHANGES({ch})      }
6:      else                          }
7:          cur.ADDCHANGES({ch_LA})   }
8:      return cur.GETCHANGES()       }
```

Alg. 5 DynaStore: traversal

```
1:  cur.ESTABLISH&COLLECTSUCCS(change)
2:      ch ← {at some s_i ∈ cur do:        }
3:          if Ch_i == { } then            } 1 round trip
4:              Ch_i ← {change}            }
5:          return some ch ∈ Ch_i          }
6:      }                                  ▷ end remote procedure
7:      cur.ADDCHANGES({ch})               } 1 round trip
8:      Ch ← cur.GETCHANGES()              }
9:      cur.ADDCHANGES(Ch)                 } 1 round trip
10:     Ch ← cur.GETCHANGES()              }
11:     cur.ADDCHANGES(Ch)   ▷ omitted if identical to Line 9
12:     return Ch
```

Figure 2 Pseudocode for reconfiguration. Client code and remote procedures invoked on servers.

be merged into a single configuration (see Figure 1b). Algorithm 5 shows how successors are established (Lines 2–7) and collected (Lines 8–11). Lines 2–6 represent a best effort approach to limit the number of successors. This was not part of the original DynaStore algorithm, but introduced in [20]. Here the reconfiguring client contacts a single server. If this server already knows of a different successor, the client will not establish a new successor. In [20] the client invokes this remote operation concurrently on a majority of the servers, but waits for only one of them to return. In our implementation, we only perform this operation on the server with lowest ID. Only if this server fails to reply, do we perform the operation on a majority of the servers. Further, if multiple clients try to register a change with a configuration C, they all try to contact the same server for Lines 2–6. Thus in the normal case, only a single successor will be established.

In DynaStore a client performs two calls of GETCHANGES to collect successors. The client also calls ADDCHANGES twice, to ensure that other clients will collect the same successors.

We refer the reader to [1, 20] for a more detailed explanation of this mechanism. In our implementation we omit the call to addChanges on Line 11 if its argument is the same as the one already used on Line 9.

Algorithm 4 shows the establishing and collecting of successors for **SM-Store**. Before calling addChanges() and establishing a new successor, a reconfiguring client proposes its changes to lattice agreement. We implement the lattice agreement algorithm from [8]. In this algorithm the client repeatedly writes and collects proposals from a majority of processes. The client adjusts its proposal, until it includes all proposals made by other clients. The change proposed to lattice agreement is included in the returned change. Further, for two changes ch_1 and ch_2 returned from lattice agreement, either ch_1 is part of ch_2 or vice versa. Thus, even though a configuration in SM-Store can have several successors, these will be ordered, e.g. all changes realized in configuration C_1 are also part of configuration C_2 (see Figure 1c). If several clients invoke lattice agreement concurrently it is likely that they all learn the same change, combining all proposed changes. Thus, the different reconfigurations will only add a single configuration to the graph. We say that the reconfigurations are batched. Pseudocode for lattice agreement can be found in [13].

To ensure that not only the successors to one configuration, but all successors are ordered, it is important that a reconfiguring client solves lattice agreement in a configuration that does not yet have a successor. Therefore, a reconfiguring client invokes getChanges on Line 3 and only if this returns an empty set, will the client use the value returned from lattice agreement to establish a new successor on Line 7. Otherwise the client enforces an existing successor (Line 5). The collection of successors is done with a simple call to getChanges on Line 8.

The **SpSn-Store** uses a speculating snapshot algorithm to both establish and collect successors. Speculating snapshot is similar to lattice agreement used in SM-Store, in that it can combine concurrently proposed changes and all established successors are ordered. Thus, the resulting graph of configurations becomes similar to SM-Store (see Figure 1c). Like SM-Store, concurrently proposed changes can also be batched using speculating snapshot. However the actual algorithm for speculating snapshot, given in [11] is quite different from lattice agreement.

A client invoking speculating snapshot performs several rounds of message exchanges, where each round has two phases. In the first phase a client disseminates its own changes and collects changes proposed by others in the same round. If no other changes are proposed, the client commits its proposal in the second phase. A committed value represents a successor configuration. If other changes have been proposed, the client disseminates all changes it has collected in the second phase. The client also collects values committed by other processes. Finally, the client starts a new round, proposing the combination of all changes observed in previous rounds. Thus in every round, at most one value is committed.

In our implementation, a client contacts a majority of the servers once for every round and phase, accessing different local variables depending on the round and phase. We refer the reader to [11] for a more thorough explanation of this algorithm. Pseudocode for the speculating snapshot algorithm can also be found in [13].

For our implementation we optimized the message pattern above using the following principle: If a call to *cur*.addChanges is always followed by *cur*.getChanges(), we simply include the local variables Ch_i in the reply returned by *cur*.addChanges. Thus, the two calls can be implemented using a single message round trip. We included braces in our pseudocode above, to show which calls are combined into one round trip. Additionally, we include the timestamped value stored at the servers in one of the message exchanges

Table 1 Differences between the studied algorithms.

	Rambo	L-Rambo	DynaStore	SpSn-Store	SM-Store
can batch concurrent reconfigurations	no	yes	no	yes	yes
round trips for establish and collect	3 to ∞	2	3 to 4	2 to $2r$	2 to $r+1$
read and *write* establish successors	no	no	yes	yes	no

performed in ESTABLISH&COLLECTSUCCS. Thus, no additional messages are necessary to collect these values on Line 10 of our template.

4.2 The Cost of a Traversal

We now perform a brief analysis of the cost of a traversal. This cost is related to the size of the successor graph that must be traversed, and the cost of establishing and collecting successor relations. We summarize this discussion in Table 1.

In Rambo, r reconfigurations representing different changes will add r configurations to the graph, since every reconfiguration creates one successor. In L-Rambo, the leader can batch these reconfigurations into fewer configurations. A stable leader can solve consensus and inform the servers about the outcome in two round-trips, thus establishing a successor. A new leader requires an additional round-trip. However, in an asynchronous system, multiple leaders may compete indefinitely for leadership and never achieve consensus [10]. Note also that a reconfiguring client may have to participate in several consensus instances, until its proposed change is chosen.

In DynaStore, r reconfigurations result in at least r new configurations. If the reconfigurations are combined in different orders by different clients, this can theoretically result in a successor graph with $2^{r-1}+r$ configurations. To traverse a single configuration normally requires only three round trips. The first of these only needs to contact a single server, not a majority. As explained above, this is because we obtain the values needed by GETCHANGES on Lines 8, 10 of Algorithm 5 from the replies of ADDCHANGES on Lines 7, 9. We omit the call to ADDCHANGES on Line 11, if its argument is the same one used on Line 9.

In SM-Store, r reconfigurations create at most r configurations, but concurrent reconfigurations may also be batched, resulting in fewer configurations. A single client requires only a single round trip to solve lattice agreement and another round trip to establish and collect successors. If r clients invoke concurrent reconfigurations, they may require r round trips to solve lattice agreement.

For SpSn-Store, r reconfigurations create at most r new configurations, possibly less. A single client can solve speculating snapshot in only two round trips, but r clients, proposing different changes concurrently, may require up to r rounds and therefore $2r$ round trips.

4.3 Read and Write Operations

In a reconfigurable storage, clients must check for successor configurations both after the query and dissemination phase of every *read* and *write* operation. If no successor configurations are found, a *read* or *write* operation precedes as in a stable system.

There are two approaches to handle a successor found during a *read* or *write* operation. In SM-Store and Rambo, a *read* or *write* operation simply traverses the graph of successor

configurations, and executing the query phase in all these configurations and similarly for the dissemination phase.

In SpSn-Store and DynaStore, when a *read* or *write* operation finds a successor, they start a reconfiguration towards this successor configuration. A *read* operation then simply returns the state collected during the traversal (v on Line 15 of the template). A *write* writes its own value to the new configuration on Line 15, instead of the collected value.

Performing a reconfiguration is clearly more costly than simply reading from or writing to all configurations in the graph. However, by performing a reconfiguration, a client ensures that any edge traversed in one operation, will not have to be traversed again by successive operations from this client. This may happen with the first approach, especially if a reconfiguring client fails while performing a reconfiguration.

5 Implementation

We have implemented all algorithms in Go 1.5 (`http://golang.org`). All algorithms implement a single register. Servers keep the algorithm state in memory and clients can only read or write the complete register. We build on a quorum RPC framework that clients use to communicate with servers. A quorum RPC sends a request to all servers in a configuration and returns after receiving replies from a quorum. Our clients always block on a quorum RPC. Our implmentation is available at `http://www.github.com/relab/smartmerge`.

Thrifty mode. Since the algorithms in our study are designed for an asynchronous system subject to failures, none of our RPCs actually require a reply from all servers in a configuration. We therefore designed a thrifty mode, where an RPC is only sent to a quorum of processes, and only after a timeout will the RPC be sent to all processes in the configuration. In our experiments, we configured this timeout to avoid resend in the absence of failures. Unless noted otherwise, all experiments are done in thrifty mode. In our implementation a client sends all thrifty RPCs to the same quorum. Different clients use different quorums to ensure that load is evenly distributed among servers.

Single contact mode. In Rambo, during the query or dissemination phase of a *read* or *write* operation, a client performs the same RPC on all configurations in the successor graph. A server's reply to this RPC in one configuration can also be used as reply in another configuration. This is possible because the servers in Rambo do not store or send information specific to a configuration. Instead, the servers send the largest timestamped value received in any configuration, and the whole successor graph, omitting only garbage collected configurations. We call this *single contact mode* (SCM) and we have implemented it for both Rambo and SM-Store. The reason SCM is also applicable to SM-Store, is that *read* and *write* operations in SM-Store are very similar to the ones in Rambo.

6 Evaluation

We evaluated the algorithms both in a local area (LAN) and wide area (WAN) setting. We evaluate both how quickly reconfigurations are applied, and the overhead these reconfigurations impose on concurrent operations.

While we use TCP for communication in all our experiments, we start servers and establish connections at startup, not during reconfigurations. This allows our evaluation to focus on the cost of running the specific reconfiguration algorithm. We believe this to be useful also

in practice, since it is possible to tell the clients to establish connections to a new server, before the reconfiguration to add this server, is actually performed.

Further, the clients in our experiments only perform *read* operations, not *write* operations, since atomic *read* operations only differ from *write*s in a small computation done locally at the client. If all values written and read have approximately the same size, then *read* and *write* operations differ neither in the kind of messages sent, nor their size.

Experiment setup. We evaluate the algorithms in a Gigabit LAN environment with machines running Linux 3.18.2. We use "small" machines with a 1.86 GHz Intel Core dual-core processor to run two servers each, and "large" machines with a 2.13 GHz Intel Xeon quad-core processor to run four clients each.

We start our experiment with an initial configuration of 8 servers each on a different "small" machine. We let 16 clients on four "large" machines continuously perform *read*s, with a payload of 4 kB. In absence of reconfigurations, these *read*s utilize the servers with more than 80 % of their maximal throughput. At one point during the experiment we start 1, 2, or 3 clients, each issuing a reconfiguration to replace one of the servers with another server, located on the same machine. Thus, every configuration actually retains the same number of servers, located on the same machines. The reconfiguring clients are located on another "large" machine. For L-Rambo we use another "large" machine to run the leader. In this setup, the leader of L-Rambo is rather over-provisioned. Initial experiments suggested, that using one of the servers in the initial configuration as leader increases reconfiguration latencies by 10-15 %, compared to the results reported below.

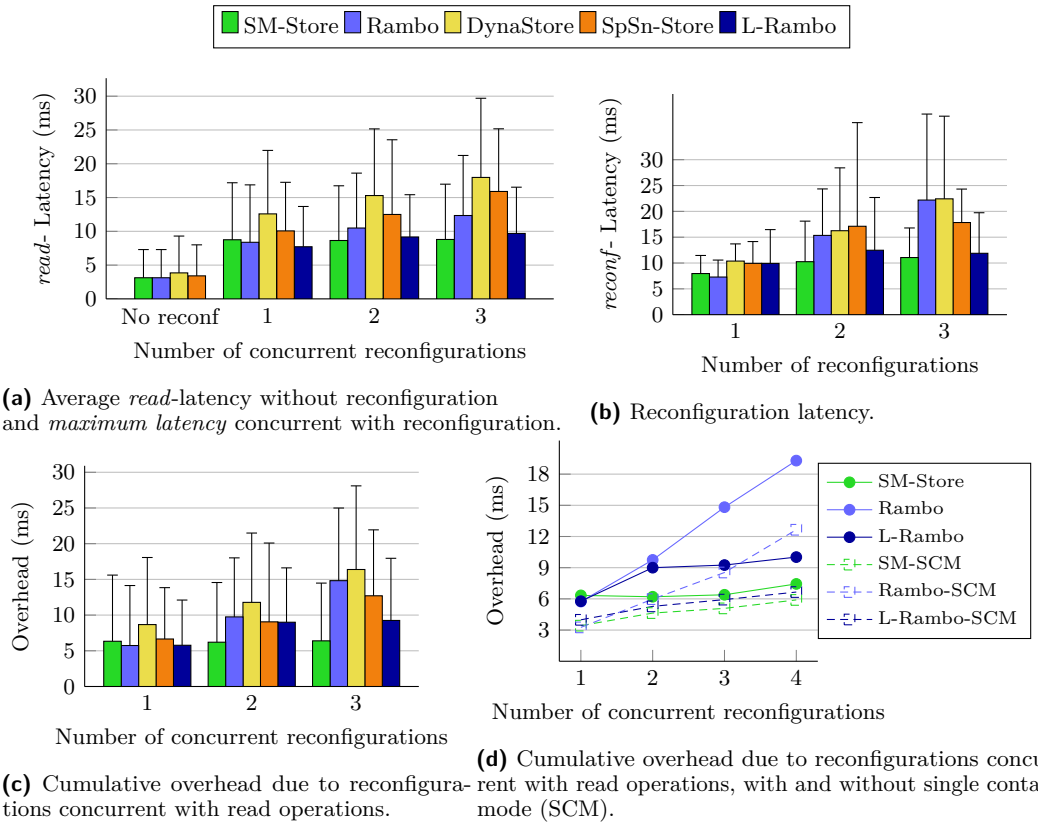
All LAN experiment are done using thrifty mode. We performed initial experiments to verify that this mode actually improves performance in all algorithms. Due to space constraints we do not report on these experiments.

Metrics. We measure both the time it takes to complete a reconfiguration and the overhead that this reconfiguration impose on concurrent read and write operations. The first measure is simply the latency of reconfiguration operations. To measure the overhead, we measure the latency of read and write operations, and label the latencies of those operations that contact several configurations. If the latency of an operation is labeled, and it is higher than the average of unlabeled latencies, we call this difference overhead.

However as we mentioned in Section 4.3, depending on how reads and writes handle successive configurations, a single reconfiguration can cause overhead to one or more operations from a client. We therefore use two metrics to evaluate the overhead. The *cumulative overhead* for a client is the total overhead that the client experienced in one run. The *maximum latency* is the maximum latency any operation from a single client experienced in one run.

While maximum latency is relevant to all clients, we believe that a small cumulative overhead is only relevant to clients that perform frequent operations.

Concurrent reconfigurations. Figure 3 shows results for handling 1-4 concurrent reconfigurations. Figure 3a shows the average read latency without reconfiguration and the **maximum read latency** a client experiences concurrent with one or more reconfigurations. The figure shows the average maximum latency and the 95th percentile for 16 clients in 40 runs. We observe that read latencies increase significantly for all algorithms during reconfiguration, but this overhead differs significantly for different algorithms. The different overhead can be explained by the characteristics we summarized in Table 1. Rambo, L-Rambo and SM-Store perform better than DynaStore and SpSn-Store since in these algorithms, *read* operations do



(a) Average *read*-latency without reconfiguration and *maximum latency* concurrent with reconfiguration.

(b) Reconfiguration latency.

(c) Cumulative overhead due to reconfigurations concurrent with read operations.

(d) Cumulative overhead due to reconfigurations concurrent with read operations, with and without single contact mode (SCM).

Figure 3 Processing a single batch of reconfigurations. Average and 95th percentile for 16 clients over 40 runs.

not establish successors, i.e. they do not complete concurrent reconfigurations. For two or more concurrent reconfigurations, SM-Store and L-Rambo perform better than Rambo, since multiple reconfiguration are batched. Thus, also fewer configurations have to be traversed by *read* operations. SpSn-Store also batches reconfigurations. but this seems to have little significance.

Figure 3b shows the actual **reconfiguration latencies**. We see again that latencies scale well for L-Rambo and SM-Store due to batching. On the other hand, latencies increase drastically for Rambo and DynaStore which do not make use of any batching.

Figure 3c shows the **cumulative overhead** due to reconfigurations. DynaStore experience the highest overhead. This is the combined effect of the lack of batching and *read* operations completing concurrent reconfigurations. The overhead of Rambo increases significantly, as more reconfigurations are invoked, due to the lack of batching. The overhead of Rambo actually grows faster than the *read* latencies for Rambo. That is because usually the initial configuration is already removed from the DAG, when the second or third configuration is added. Thus no single *read* operation needs to contact all configurations. SM-Store scales well since even three reconfigurations are batched into a single new configuration. In L-Rambo, only the second and third reconfiguration are batched. This is because we do not use any batching timeout, but instead propose the first reconfiguration immediately.

We now evaluate the **single contact mode**, where clients try to avoid contacting the same process in different configurations. We have implemented this mode for Rambo, L-Rambo, and SM-Store, as explained in Section 5. The experiment setup is the same as

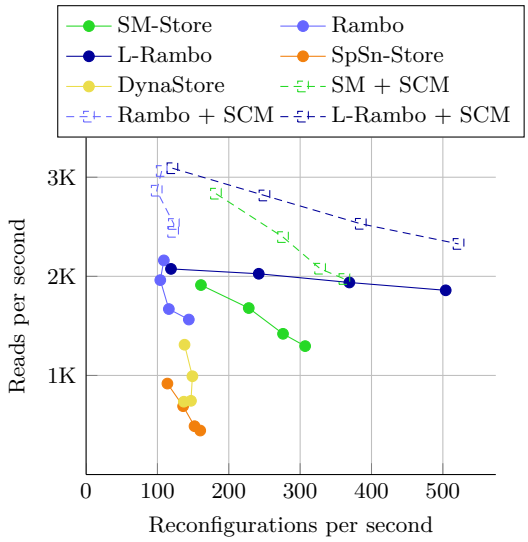


Figure 4 Read and reconfiguration throughput with 2, 4, 6, and 8 reconfiguring clients, each represented by a dot or square in the graphs. More clients increases reconfiguration throughput. Average over 20 runs, each with a 30 second duration.

in the previous experiment. Figure 3d shows that cumulative overhead, with and without single contact mode, in scenarios with 1, 2, 3, and 4 concurrent reconfigurations. We see that, while its effect on SM-Store is limited, single contact mode has a significant impact on L-Rambo, mitigating the difference between L-Rambo and SM-Store. For Rambo without leader, single contact mode partially mitigates the lack of batching. However, for a larger number of concurrent reconfigurations (e.g. 4), Rambo still experiences significantly larger overhead than the other variants.

We repeated the above experiments using **regular reads** that omit the dissemination phase from *read* operations. This experiment also serves as an estimate for optimizations that maintain atomic semantics, but omit the dissemination phase when possible. Such optimizations (e.g. RDS [5]) are equally applicable to Rambo, L-Rambo and SM-Store. Omitting the dissemination phase from *read* operations reduces normal case *read* latencies by 66% for SM-Store and Rambo and 49% for DynaStore and SpSn-Store. Maximum latencies during reconfiguration are also reduced by 40-50% in SM-Store, Rambo and L-Rambo. But *read* operations that complete concurrent reconfigurations (DynaStore and SpSn-Store) maintain the same latencies. Operations that complete concurrent reconfigurations must disseminate values to the new configuration. Thus the dissemination phase cannot be completely omitted in DynaStore and SpSn-Store.

Measurements on overhead, reconfiguration latency and the impact of single contact mode are qualitatively similar to the ones reported above.

Constant reconfiguration. We are also interested in the questions: what frequency of reconfigurations can the algorithms support, and how does a constant rate of reconfigurations impact *read* throughput?

Under constant reconfiguration, the algorithms cannot guarantee that operations complete. That is because reconfigurations might be adding configurations to the graph faster than a *read* operation can traverse this graph. However, our experiment shows that several algorithms can still maintain reasonable throughput.

We use 8 servers and 16 *read*-clients as in the previous experiment. We then start 2, 4, 6, and 8 clients that continuously replace different servers, switching back-and-forth between two servers located on the same machine.

Figure 4 plots the number of completed reconfigurations against the number of completed read operations per second. We see that Rambo, SM-Store, and especially L-Rambo maintain reasonable read throughput.

For most algorithms, additional reconfiguring clients result in more completed reconfigurations. DynaStore, SpSn-Store, and Rambo all complete between 100 and 160 reconfigurations per second. However, in Rambo significantly more *read* operations complete concurrent with these reconfigurations than in DynaStore and SpSn-Store. That is, because *read* operations in Rambo do not complete concurrent reconfigurations. In DynaStore, adding additional reconfiguring clients does not improve reconfiguration throughput but still reduces *read* throughput. That is because in DynaStore all reconfigurations and *read* operations that complete concurrent reconfigurations try to contact a single server to establish a successor configuration (see Section 4). In this experiment, this single server becomes a bottleneck.

Figure 4 also shows results for single contact mode. This mode significantly improves the read throughput but has little effect on the number of completed reconfigurations.

6.1 WAN experiments

In this section we present experiments performed in a wide area network (WAN).

We performed similar experiments to those reported above, using Amazon Web-Services micro instances running Ubuntu 14.04 in several data centers. We used a different instance for each client and server.

In our experiments we started with a configuration with 3 servers and 3 clients continuously performing *read* operations. A client and a server were located in each of Europe (Frankfurt), US West (N. California) and Asia (Tokyo). The *read* operations have a payload of 100 bytes. For L-Rambo we use an additional instance located in US West as leader.

We did not use thrifty mode in these experiments, because using this mode in a wide area network requires the client to carefully choose the servers with the lowest latency. This is especially difficult to determine for servers in new configurations where a client cannot rely on the latencies from previous requests.

We first evaluate the algorithms under **constant reconfiguration**. For this experiment we use three clients, located in each of the above mentioned data centers. During 30 runs each lasting 60 seconds, the clients constantly propose reconfigurations. Every reconfiguration proposes to replace the server located in the same data center as the reconfiguring client, with a server, located on another instance in the same data center. Figure 5 shows average latencies for these *reconf* operations and concurrent *read*s and the 95th percentile. Note that the two measurements must be seen in conjunction. For example, since reconfigurations in Rambo often take several seconds to complete, only a few reconfigurations actually complete in a run. Few reconfiguration only cause a small overhead to *read* operations.

The *reconf* latencies for Rambo, SM-Store are dominated by extreme spike latencies. SpSn-Store also experiences some extreme spike latencies, which may exceed the experiment duration (> 60 seconds). While one client experiences a spike latency, another client may preform many reconfigurations. Thus spike latencies form less than 5% of the observed latencies and have little effect on the 95th percentile, but cause the average to lie above this percentile.

The high spike latencies for reconfigurations all come from the reconfiguring client in Europe. In Rambo and SpSn-Store, in some runs, this client does not manage to successfully

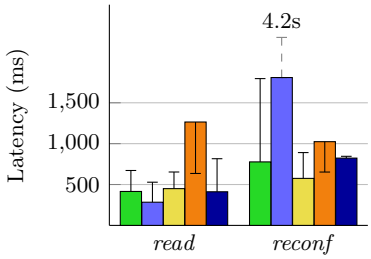
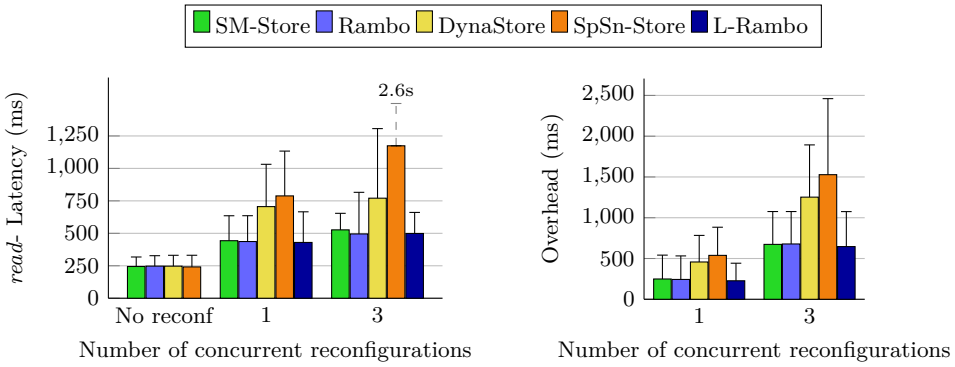


Figure 5 Latencies in WAN under constant reconfiguration. Average and 95th percentile for 30 runs lasting 60 seconds each.



(a) Read latency in WAN without, with 1 or with 3 concurrent reconfigurations.

(b) Overhead in WAN with 1 or 3 concurrent reconfiguration.

Figure 6 Latencies in WAN. Average and 95th percentile obtained from 30 runs.

apply its changes, before the end of the experiment. In SpSn-Store this also happened to the European client performing reads.

In SM-Store the reconfiguring client from Europe always manages to complete a request during the experiment, but requires up to 30 seconds to do so. We see that in SpSn-Store these spike latencies extend to *read* operations, since also *read* operations participate in the Speculating snapshot. In Rambo and SM-Store on the other hand, the spike latencies for reconfigurations have little impact on the *read* latencies.

Surprisingly, in this experiment DynaStore performs especially well, with the lowest average reconfiguration latency of all algorithms in our study, and an average read latency that is similar to the other algorithms. As described in Section 4, DynaStore uses one of the servers in the configuration to prevent multiple successors. In this experiment, this server is located in Europe, which gives an advantage to the reconfiguring client located in Europe. It is this client that experiences spike latencies in the other algorithms.

We also measured the latency and overhead of a **single batch of reconfigurations** in our wide area setting. In this experiment, we first let a single client propose a reconfiguration, replacing one server with a new one, located in the same data-center. We alternate both on the location of the client and, which server is reconfigured. We also performed an experiment, where all three reconfiguring clients, one in each data center, concurrently perform one reconfiguration each. However since the three clients are separated by significant latencies, the reconfigurations are not as closely synchronized as in the LAN experiments, where all reconfiguring clients where located on the same machine.

Figure 6a shows normal and maximum read latency for this experiment. Figure 6b shows the overhead caused by 1 or 3 concurrent reconfigurations. Since the reconfigurations are not closely synchronized, the batching mechanisms fails to combine them. Thus, SM-Store, L-Rambo perform similar to Rambo. SpSn-Store performs worth than DynaStore for both a single, and concurrent reconfigurations. This is caused by the batching mechanism in SpSn-Store, which is performed by all clients, but has little effect in the WAN setting.

7 Conclusion

We have evaluated different algorithms for reconfiguration of atomic storage, both with and without consensus. For the different algorithms, we measure both reconfiguration latencies and the overhead caused by reconfigurations. Our experiments show that novel algorithms for reconfiguration without consensus perform similar to consensus-based L-Rambo, if the latter has a stable leader. However, especially SM-Store performs significantly better than Rambo, when the latter is run without a stable leader. Our experiments suggest that if *read* and *write* operations do not help concurrent reconfigurations to complete, that significantly reduces the overhead.

References

1 Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2), 2011.

2 Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI 2014*.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869.

4 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC'15, pages 241–250, New York, NY, USA, 2015. ACM.

5 Gregory Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alex A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing*, 69(1), 2009.

6 Gregory Chockler, Dahlia Malkhi, and Danny Dolev. A data-centric approach for scalable state machine replication. In André Schiper, AlexA. Shvartsman, Hakim Weatherspoon, and BenY. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 159–163. Springer Berlin Heidelberg, 2003.

7 Partha Dutta, Rachid Guerraoui, Ron R Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM Journal on Computing, Vol 39, N°8, December 2010*, 2010.

8 Jose M. Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC 2012*, pages 125–134. ACM, 2012.

9 Rui Fan and Nancy Lynch. Efficient replication of large data objects. In Faith Ellen Fich, editor, *Distributed Computing: 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003. Proceedings*, pages 75–91. Springer, 2003.

10 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

11 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In Yoram Moses, editor, *Distributed Computing: 29th International Conference, DISC 2015. Proceedings*, pages 140–153. Springer, 2015.

12 S. Gilbert, N. Lynch, and A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distr. Comp.*, 23(4), 2010.

13 L. Jehl and H. Meling. Additional material. URL: `http://www.ux.uis.no/~ljehl/pdf/thecase.pdf`.

14 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In Yoram Moses, editor, *Distributed Computing: 29th International Conference, DISC 2015. Proceedings*, pages 154–169. Springer, 2015.

15 Leslie Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

16 Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

17 Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *EuroSys*, 2006.

18 Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SOCC*, 2012.

19 Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multi-writer consistency conditions for the shared memory objects. In *DISC 2003*.

20 Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *LADIS 2010*.

21 Daniel Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.

Step Optimal Implementations of Large Single-Writer Registers*

Tian Ze Chen[1] **and Yuanhao Wei**[2]

1 **Department of Computer Science, University of Toronto, Toronto, Canada**
`tianze.chen@mail.utoronto.ca`
2 **Department of Computer Science, University of Toronto, Toronto, Canada**
`yuanhao.wei@mail.utoronto.ca`

Abstract

We present two wait-free algorithms for simulating an ℓ-bit single-writer register from k-bit single-writer registers, for any $k \geq 1$. Our first algorithm has $\Theta(\ell/k)$ step complexity for both READ and WRITE and uses $\Theta(4^{\ell-k})$ registers. An interesting feature of the algorithm is that READ operations do not write to shared variables. Our second algorithm has $\Theta(\ell/k + (\log n)/k)$ step complexity for both READ and WRITE, where n is the number of readers, but uses only $\Theta(n\ell/k + n(\log n)/k)$ registers. Combining both algorithms gives an implementation with $\Theta(\ell/k)$ step complexity using $\Theta(n\ell/k)$ space for any $1 \leq k < \ell$.

We also prove that any implementation with $O(\ell/k)$ step complexity for READ requires $\Omega(\ell/k)$ step complexity for WRITE. Since reading ℓ-bits requires at least $\lceil \ell/k \rceil$ reads of k-bit registers, our lower bound shows that our implementation is step optimal.

1998 ACM Subject Classification E.1 Distributed Data Structures, F.1.2 Parallelism and Concurrency

Keywords and phrases atomic register, regular register, wait-free implementation, single writer, optimal

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.32

1 Introduction

A register is a fundamental object that supports READ and WRITE operations. Implementing large ℓ-bit registers from small k-bit registers in a wait-free manner is a classic problem in distributed computing. We consider this problem for atomic single-writer registers shared by n readers. This problem arises naturally in practice when ℓ-bits need to be written atomically on a system that provides only k-bit single-writer registers.

We define the *space complexity* of an implementation of ℓ-bit registers from shared k-bit registers to be the number of k-bit registers that it uses, and we define the *step complexity* to be the number of **read** and **write** operations on these k-bit registers. Note that $\lceil \ell/k \rceil$ **read** steps are required for an ℓ-bit READ operation. Also, any implementation requires $\lceil \ell/k \rceil$ space, since 2^ℓ different values need to be represented.

In 1983, Peterson [9] presented an implementation with $\Theta(\ell/k)$ step complexity for both READ and WRITE. Peterson's implementation has space complexity $\Theta(n\ell/k)$ and works for all $k \geq 1$.

Later, Larsson et al. [8] improved the step complexity of WRITE to $\Theta(n + \ell/k)$, but they used SWAP and FETCHANDOR as primitives.

* This work was supported by the Natural Science and Engineering Research Council of Canada.

Algorithm	Read	Write	Space	Restriction
Peterson (1983)	$\Theta(\ell/k)$	$\Theta(n\ell/k)$	$\Theta(n\ell/k)$	None
Chaudhuri, Kosa, Welch (2000)	$\Theta(4^\ell)$	1	$\Theta(4^\ell)$	None
Aghazadeh, Golab, Woelfel (2014)	$\Theta(\ell/k)$	$\Theta(\ell/k)$	$\Theta(n\ell/k)$	$k \in \Omega(\log n)$
This Paper	$\Theta(\ell/k)$	$\Theta(\ell/k)$	$O(n\ell/k)$	None

Figure 1 Step complexities for implementations of an atomic ℓ-bit single-writer register from atomic k-bit single writer registers.

In 1991, Vidyasankar [10] showed that an atomic ℓ-bit register can be implemented from two regular ℓ-bit registers and one atomic binary register. His WRITE algorithm performs 2 regular writes and 2 atomic writes, and his READ operation performs 2 regular reads and 1 atomic read in the worst case.

Later, Chaudhuri and Welch [4] presented an implementation of regular ℓ-bit registers from regular binary registers with step complexity $\Theta(\ell)$ for both READ and WRITE, using $\Theta(2^\ell)$ space.

In 2000, Chaudhuri, Kosa and Welch [3] presented an atomic ℓ-bit register implementation from atomic binary registers in which each WRITE operation performs a single step. However, the step complexity of their READ operation and their space complexity are both $\Theta(4^\ell)$.

Recently, Aghazadeh, Golab and Woelfel [1] implemented an ℓ-bit *multi-writer* register from k-bit *multi-writer* registers with step complexity $\Theta(\ell/k)$ for both READ and WRITE. Their implementation uses $\Theta(n^2\ell/k)$ registers and requires that $k \in \Omega(\log n)$.

The table in Figure 1 summarises the existing implementations. Peterson's, Chaudhuri and Welch's, and Aghazadeh, Golab and Woelfel's implementations are described in more detail in Section 3. Also in Section 3, we show how to modify Aghazadeh, Golab and Woelfel's implementations to obtain an implementation of an ℓ-bit *single-writer* register from k-bit *single-writer* registers with $\Theta(\ell/k)$ step complexity and $\Theta(n\ell/k)$ space complexity, provided that $k \in \Omega(\log n)$.

In this paper, we present an implementation of an atomic ℓ-bit single-writer register from atomic k-bit single-writer registers with $\Theta(\ell/k)$ step complexity that works for all $k \geq 1$. Our implementation uses $O(n\ell/k)$ registers, which is the same as Peterson's implementation and the single-writer variant of Aghazadeh et al.'s implementation.

We show that our implementation is optimal by proving that any implementation with $O(\ell/k)$ step complexity for READ requires $\Omega(\ell/k)$ step complexity for WRITE.

Our register implementation is the composition of a *tree based* implementation and a *buffer based* implementation. Our tree based implementation has $\Theta(\ell/k)$ step complexity and uses $\Theta(4^{\ell-k})$ registers. An interesting feature is that readers never write to shared registers. This means that helping techniques, such as announcing operations and handshaking, are not used. This implementation can be modified to implement a modulo m counter from k-bit *single-writer* registers that supports a single incrementer and any number of readers. Our counter uses $\Theta(m/2^k)$ registers and has $\Theta((\log m)/k)$ step complexity for READCOUNTER and INCREMENT.

Our buffer based implementation uses this counter as well as known techniques, such as announcement arrays, round-robin helping, and handshake objects, to obtain an implementation with step complexities $\Theta(\ell/k + (\log n)/k)$, while using only $\Theta(n\ell/k + n(\log n)/k)$ registers.

When $\ell \leq \lceil(\log_2 n)/2\rceil$, our tree based implementation has optimal step complexity and uses $O(n/4^k)$ registers. When $\ell > \lceil(\log_2 n)/2\rceil$, our buffer based register implementation has

optimal step complexity and uses $\Theta(n\ell/k)$ registers. Combining these two algorithms gives a step optimal implementation using $O(n\ell/k)$ registers for any $1 \leq k < \ell$.

2 Preliminaries

A *single-writer register* R is a shared register where only one process can perform WRITE operations and any number of processes can perform READ operations. We say that a process owns R if it can write to R.

A *single-incrementer modulo m counter* is a shared modulo m counter where only one process can perform INCREMENT operations and any number of processes can perform READCOUNTER operations. We say that a process owns the counter if it can increment the counter. The counter can take on values from 0 to $m-1$.

We will work in the standard asynchronous shared memory model with n readers p_0, p_1, $\ldots$, p_{n-1} and one writer, which communicate through k-bit registers. Processes may fail by crashing under our model.

In our model, an *execution* is an alternating sequence of *configurations* and *steps* C_0, e_1, C_1, e_2, C_2, $\ldots$, where C_0 is an *initial configuration*. Each step is either a **read** or **write** of a k-bit register. Configuration C_i consists of the state of every register and every process after the step e_i is applied to configuration C_{i-1}. For any two configurations C and C', we use $C \to C'$ to denote that C precedes C' in the execution.

If $C \to C'$, the *execution interval* $[C, C']$ is the set of all configurations and steps between C and C', inclusive. Similarly, the *execution interval* of an operation is the set of all configurations and steps from the first step of that operation to the configuration immediately after the last step of that operation. The execution interval for an *incomplete operation* is the set of all configurations and steps starting from the first step of that operation. Two execution intervals *intersect* if they have a common configuration or step.

We say an object is *atomic* (or equivalently, its implementation is *linearizable* [5]) if, for every possible execution and for each operation on that object in the execution, we can pick a configuration in its execution interval to be its linearization point, such that the operation appears to occur instantaneously at this point. In other words, all operations on the object must behave as if they were performed sequentially, ordered by their linearization points.

We say a register is *regular* if the value returned by each READ is either the value written by the last WRITE operation completed before the first step of the READ or the value written by a WRITE operation concurrent with the READ operation. Note that every atomic register is also regular.

To emphasize the important distinction between atomic and regular registers, consider the scenario where a WRITE operation is concurrent with two READ operations by the same process. Suppose the WRITE operation changed the register from value a to value b. If the register is regular, then each read operation is allowed to return either a or b. However, if the register is atomic, then the second read operation must return b if the first READ operation returns b.

All implementations that we discuss will be *wait-free*. This means that each operation by any process p_i is guaranteed to complete within a finite number of steps by p_i. The *step complexity* of an operation O is the maximum number of steps, over all possible executions, that the process which invoked O performs in the execution interval of O. The *space complexity* of an implementation is defined to be the number of shared registers that it uses.

3 Related Work

In this section, we will briefly review a few existing implementations. Our implementation will build upon these implementations.

Both Peterson's and Aghazadeh et al.'s implementations represent an ℓ-bit value using an array of $\lceil \ell/k \rceil$ registers, each containing k-bits. This construction is called a *buffer*.

In Peterson's implementation, there are two global buffers, $G[0]$ and $G[1]$. Each reader also has its own message buffer. WRITE operations alternate between writing their values to $G[0]$ and $G[1]$. The writer flips a binary register V to indicate the currently active buffer. After flipping V, the writer then checks for help requests from each reader and writes the current value into the message buffers of the readers that requested help. Finally, the writer acknowledges those help requests. At a high-level, a READ operation performs the following steps: request help from the writer, read $G[V]$, and check for an acknowledgement. If no acknowledgement was received, then the value that it read from $G[V]$ has not been overwritten, so it returns what it read. Otherwise it uses the value that the writer put in its message buffer. Essentially, the idea is that if the reader is fast, it can return what it read from G, and if the reader is slow, it will be helped by the writer. Peterson's implementation has step complexities $\Theta(\ell/k)$ and $\Theta(n\ell/k)$ for READ and WRITE respectively, and space complexity $O(n\ell/k)$.

Aghazadeh et al.'s implementation is more complex and requires a novel garbage collection scheme, which they introduced. They implemented a large *multi-writer* register from small *multi-writer* registers. Their implementation achieves $\Theta(\ell/k)$ step complexity for WRITE by having each writer help readers in a round-robin fashion. Thus, only one reader is helped during each WRITE. Since helping is less frequent in their algorithm, they use an array G of $n(8n+1)$ buffers, instead of 2 buffers, to ensure that a reader is helped before the value it wants to read is overwritten. This means that they require $\Theta(\log n)$ size registers to store pointers to elements in G, which leads to the requirement that $k \in \Omega(\log n)$.

Their implementation can be converted into an implementation of an ℓ-bit single-writer register from k-bit single-writer registers. When there is only one writer, each multi-writer register in their implementation is shared by only two processes. Israeli and Shaham [6] presented an implementation of a multi-writer register shared by p processes from single-writer registers of the same size with $\Theta(p)$ step complexity for READ and WRITE. We can use this implementation to simulate the multi-writer registers in Aghazadeh et al.'s implementation from single-writer registers in constant time and constant space. This results in an implementation of an ℓ-bit single writer register from k-bit single writer registers which works for $k \in \Omega(\log n)$ and has $\Theta(\ell/k)$ step complexity. In this case, G only needs to contain $\Theta(n)$ buffers, so the space complexity is reduced to $\Theta(n\ell/k)$.

Now we turn our attention to Chaudhuri and Welch's regular register implementation. They use a complete binary tree where each leaf represents a different register value and each internal node stores a *switch*, a shared binary regular register that selects between its two children. Their regular register read operation traverses down the tree, following the switches, until it reaches a leaf and returns the value of that leaf. Their regular register write operation starts at the leaf with the value it wishes to write and traverses up the tree, changing each switch on its path to point towards that leaf.

Using Chaudhuri and Welch's implementation for the regular ℓ-bit registers in Vidyasankar's algorithm gives an implementation of an atomic ℓ-bit register from $\Theta(2^\ell)$ regular binary registers and one atomic binary register with $\Theta(\ell)$ step complexity. We call this the CWV implementation. The CWV implementation can be used in place of our tree

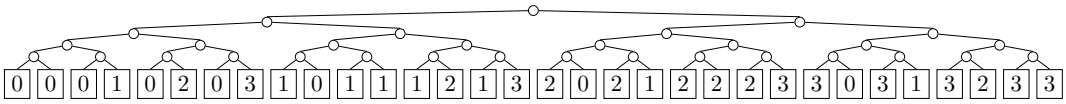


Figure 2 Atomic 4-valued register.

based implementation in our final optimal implementation. However, our implementation of a counter, which is based on our tree based implementation, is faster than the CWV implementation by a factor of 2. This counter will be used in our buffer based implementation.

4 Tree Based Implementation

We begin by showing how Chaudhuri and Welch's regular register implementation can be extended to an atomic register implementation. Then we show how both these implementations can be generalized to work for $k > 1$. In this section, it is more natural to talk about m-valued registers, where $m = 2^\ell$, rather than ℓ-bit registers.

4.1 Implementing an m-valued atomic register

Consider Chaudhuri and Welch's regular register implementation where each binary switch is atomic rather than regular. Notice that the WRITE operation in this implementation is atomic in the case where only one switch is changed on the path from the leaf to the root. This observation is the key behind our atomic register implementation. To construct an atomic register, we use a larger tree such that, for every pair of values, there are two leaves with these values that have a common parent. This allows us to change from the current value to any new value by changing only one switch. $\binom{m}{2}$ height 1 nodes are needed to guarantee this property, but we use m^2 height 1 nodes to simplify the implementation.

More formally, we construct a complete, perfectly balanced binary tree with m^2 height 1 nodes, $w_0, w_1, ..., w_{m^2-1}$. Each internal node stores a shared binary register called a switch which selects between its two children. All the binary registers can be regular except for those in the height 1 nodes, which must be atomic. Figure 2 illustrates an atomic 4-valued register.

The node w_i has a left child with value $\alpha = \lfloor i/m \rfloor$ and a right child with value $\beta = (i \bmod m)$. So each pair of values (α, β) has a common parent $w_{\alpha * m + \beta}$.

Algorithm 1 presents the pseudo-code for Chaudhuri and Welch's implementation, which we will use as a subroutine. A REGULARWRITE(i) operation changes the switches in the tree to point towards the leaf with index i. The fields of each node are immutable except for *switch*. The variable *root* and the contents of the array *leaves* are also immutable, and *node* is a local variable. Immutable variables and fields can be stored in the local memory of each process, instead of being stored in shared memory.

Our atomic READ algorithm starts at the root and returns the value of the leaf that it arrives at by following switches, just like REGULARREAD.

Our atomic WRITE(val) operation first computes the height 1 node, *parent*, whose left child has the current value and whose right child has the value being written. Then it performs REGULARWRITE with the index of its left child. Next, it changes *parent*'s switch to point to its right child. This step is exactly the same as performing a REGULARWRITE with the index of its right child, because all other switches on the path to the root remain the same. The WRITE operation is linearized immediately after this step.

Algorithm 1 Chaudhuri and Welch's implementation of a regular m-valued register.

```
0: procedure RegularRead()
1:     node ← root
2:     while node is not a leaf do
3:         s ← read(node.switch)
4:         if s = 0
5:         then node ← node.left
6:         else node ← node.right
7:     return node.value
```

```
0: procedure RegularWrite(index)
1:     node ← leaves[index]
2:     while node is not the root do
3:         if node is a left child
4:         then write(node.parent.switch, 0)
5:         else write(node.parent.switch, 1)
6:         node ← node.parent
```

Algorithm 2 Implementation of an atomic m-valued register.

```
0: procedure Read()
1:     return RegularRead()
```

```
0: procedure Write(val)
1:     parent ← parent(oldval, val)
2:     RegularWrite(parent.left.index)
3:     write(parent.switch, 1)
4:     oldval ← val
```

Pseudo-code for our atomic register implementation is presented in Algorithm 2. In the pseudo-code, *oldval* is a persistent local variable that is initialized to the initial value of the register. The function **parent**$(oldval, val)$ is performed locally and returns the height 1 node whose left child has value *oldval* and whose right child has value *val*. Both Read and Write operations take $\Theta(\log m)$ steps. This immediately implies that they are wait-free.

Fix an execution of Read and Write operations. The variables are initialized so that it appears as if a complete Write of 0 has occurred before any Read operation. Let R be a Read operation in this execution which returns α. If there is a Write of α linearized in the execution interval of R, then linearize R immediately after the linearization point of the first such Write operation. In this case, R returns the value of the last Write operation linearized before it.

Now suppose there is no Write of α linearized in the execution interval of R. In this case, we can linearize R immediately after its first step. We need to show that the last Write operation W linearized before the first step of R is a Write of α.

Since W is linearized immediately after its last step, W completes before R begins. If the writer does not start another Write until after R completes, then R is concurrent with no RegularWrite operations and, hence, returns the value written by W. So, suppose the writer starts another Write before R completes. Let W' be the first Write operation following W.

Suppose, for contradiction, that W returns $\beta \neq \alpha$. Since W' is linearized at line 3, line 3 of W' occurs after the first step of R by definition of W. Notice that a Write operation can be viewed as two RegularWrite operations. This is because the atomic **write** performed on line 3 is equivalent to RegularWrite(*parent.right.index*), since the writer performed RegularWrite(*parent.left.index*) immediately beforehand.

Therefore the last complete RegularWrite operation before the start of R is either the second RegularWrite of W or the first RegularWrite of W'. From the code, we can see that both operations write the value β.

R consists of a single RegularRead, so by the correctness of Algorithm 1, there exists a RegularWrite of value α concurrent with R. The first such RegularWrite operation must be the second RegularWrite of some Write operation W''. This is because the first RegularWrite of each Write operation writes the same value as the second

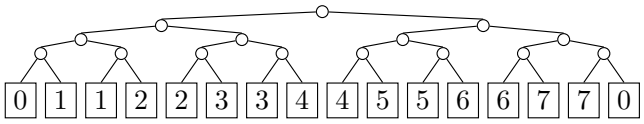


Figure 3 Atomic 8-valued counter.

RegularWrite of the previous Write operation. Since the second RegularWrite of W'' is atomic and W'' is linearized at the following configuration, W'' is linearized in the execution interval of R which contradicts our assumption. Therefore W must have written α.

▶ **Theorem 1.** *Algorithm 2 implements an atomic m-valued register.*

4.2 Implementing a Modulo m Counter

It is easy to modify our atomic m-valued register implementation to implement an atomic modulo m counter. Since the value can only be incremented, we only need one height 1 node for each $(\alpha, (\alpha+1) \bmod m)$ pair. Hence the space complexity can be reduced to $\Theta(m)$. ReadCounter performs the same steps as the atomic register Read, and Increment is the same as Write. Figure 3 illustrates an atomic 8-valued counter.

4.3 Extension to k-bit registers

To extend Chaudhuri and Welch's implementation and our atomic counter implementation to use k-bit registers, for $k > 1$, we replace the binary tree in each implementation with a 2^k-ary tree, and keep the same set of leaves. This simple modification reduces the height of the tree to $\Theta(\log_{2^k}(m))$, so that all operations have step complexity $\Theta((\log m)/k)$. We use one k-bit register for each internal node so the space complexity is equal to the number of internal nodes. Since there are $\Theta(m/2^k)$ height 1 nodes, there are $\Theta(m/2^k)$ internal nodes and the space complexity is also $\Theta(m/2^k)$.

For our atomic register implementation, we partition the values into sets of size 2^{k-1}, and have one height 1 node for each pair of these sets. This construction ensures every pair of values are siblings in the tree. Thus, we only need $(m/2^{k-1})^2$ height 1 nodes. Therefore, the step complexity becomes $\Theta((\log m)/k)$ and the space complexity becomes $\Theta(m^2/4^k)$. An m-valued register is the same as an ℓ-bit register when $m = 2^\ell$, so the step complexity is $\Theta(\ell/k)$ and the space complexity is $\Theta(4^{\ell-k})$. If $\ell \leq \lceil(\log_2 n)/2\rceil$, we have $m \leq 2\sqrt{n}$, which means the implementation has space complexity $O(n/4^k)$.

5 Buffer Based Implementation

We begin by describing the handshake object, a primitive that we use in our implementation. In Section 5.2, we describe our buffer based implementation and analyse its step and space complexities. Then, in Section 5.3, we introduce notation that is used in its correctness proof. Some important handshaking properties are proven in Section 5.4. The proof of correctness appears in Section 5.5.

5.1 Handshaking

A handshake object H is used to coordinate between pairs of processes and can be implemented using a pair of 1-bit single-writer registers $H.r$ and $H.w$. We will use a handshake object H_i

in the buffer based implementation to coordinate between the reader p_i and the writer. The idea of handshaking first appeared in papers by Peterson [9] and Lamport [7]. The version we will use is from Attiya and Welch [2].

We say that reader p_i *requests help* when it sets $H_i.r$ equal to the value it read from $H_i.w$. The writer *acknowledges a help request from* p_i when it sets the value of $H_i.w$ to be the opposite of what it read from $H_i.r$. The reader p_i *checks for an acknowledgement* by checking if these two handshake bits are different and the writer *checks for a help request from* p_i by checking if these two handshake bits are the same.

Intuitively, the handshake object guarantees that a check for an acknowledgement by p_i returns true if and only if the writer has acknowledged a help request from p_i since p_i's last help request. Similarly, a check for a help request from p_i by the writer returns true if and only if there has been a help request by p_i since the writer's last acknowledgement to p_i.

5.2 Description

As in Peterson's implementation, our buffer-based implementation uses an array of buffers G and a pointer V to the currently active buffer in G. However, G contains $4n$ buffers, instead of 2, and V is implemented using a single-incrementer modulo $4n$ counter, as described in Section 4.2. Like Aghazadeh, Golab and Woelfel's implementation, our implementation uses round-robin helping, except that it uses handshaking and completion bits to coordinate the helping.

In round-robing helping, each WRITE operation only helps a single reader. If the current WRITE operation helps process p_i, then the next WRITE operation will help process $p_{(i+1) \bmod n}$. This ensures that each reader p_i is helped once every n WRITE operations.

A reader begins by reading the counter V and announcing that value. This value is the index of the element in G that it tries to read. A WRITE operation that sees the announcement helps the reader by sending it the requested element of G, as well as the value that it just wrote. Reader p_i announces the value it read from V using an array A_i of size $\lceil (logn)/k \rceil$. It is the only process that writes to this array and only the writer reads from this array. The writer uses buffers M_i and N_i to send the value that was requested and the value it just wrote, respectively, to p_i. Only the writer can write to these two buffers, and only the reader p_i can read from these two buffers. The arrays A_i, M_i and N_i are accompanied by a completion bit which is set and cleared only by the process that writes to the array. The completion bit is set after a complete write to the array and while the completion bit is set, the array will not be written to. The completion bit of A_i is reset at the beginning of a READ$_i$ operation, and the completion bits of M_i and N_i are reset when the writer notices a new help request from p_i.

A WRITE(v) operation first computes the reader p_i that it should help and checks for a help request from that reader. If there is no help request from p_i, the writer writes v into buffer $G[(V+1) \bmod 4n]$ and increments V to point to this buffer. If there is a help request, the writer clears the completion bits of N_i and M_i, and acknowledges the help request. Next it writes v into buffers $G[(V+1) \bmod 4n]$ and N_i, increments V, and sets the completion bit of N_i. In either case, if the completion bit of M_i is not set and the completion bit of A_i is set, the writer copies the contents of buffer $G[A_i]$ into M_i and sets the completion bit of M_i.

A READ operation by reader p_i first clears the completion bit of its announcement array A_i and requests help. Then it announces the current value of V and sets the completion bit of A_i. If the writer has sent an acknowledgement and the completion bit of N_i is set, the reader returns the value in N_i. Otherwise, the reader reads buffer $G[A_i]$ and performs another check; if the writer still has not sent an acknowledgement or if the completion bit of

Algorithm 3 Buffer based implementation of an ℓ-bit register from k-bit registers.

$G[0 \dots 4n-1]$: array of buffers
V: modulo $4n$ counter
For each $i = 0, \dots, n-1$:
  N_i: buffer
  M_i: buffer
  F_{N_i}, F_{M_i}: completion bits
  $H_i.w$: writer's handshaking bit
  A_i: array of $\lceil (\log 4n)/k \rceil$ register
  F_{A_i}: completion bit
  $H_i.r$: reader's handshaking bit

```
0:  procedure READ_i()
1:      write(F_{A_i}, 0)
2:      t ← read(H_i.w)
3:      write(H_i.r, t)
4:      version ← READCOUNTER(V)
5:      writearray(A_i, version)
6:      write(F_{A_i}, 1)
7:      h ← H_i.r ≠ read(H_i.w)
8:      if h ∧ read(F_{N_i}) then
9:          val ← readarray(N_i)
10:     else
11:         val ← readarray(G[version])
12:         h ← H_i.r ≠ read(H_i.w)
13:         if h ∧ read(F_{M_i}) then
14:             val ← readarray(M_i)
15:     return val
```

```
0:  procedure WRITE(value)
1:      version ←
            (READCOUNTER(V) + 1) mod 4n
2:      i ← version mod n
3:      if read(H_i.r) = H_i.w then
4:          write(F_{N_i}, 0)
5:          write(F_{M_i}, 0)
6:          t ← read(H_i.r)
7:          write(H_i.w, 1 − t)
8:      writearray(G[version], value)
9:      INCREMENT(V)
10:     if ¬F_{N_i} then
11:         writearray(N_i, value)
12:         write(F_{N_i}, 1)
13:     if ¬F_{M_i} ∧ read(F_{A_i}) then
14:         rversion ← readarray(A_i)
15:         writearray(M_i, G[rversion])
16:         write(F_{M_i}, 1)
```

M_i is not set, the reader returns the value it read from $G[A_i]$. Otherwise, the writer has sent an acknowledgement and M_i contains the value that was requested, $G[A_i]$, so the reader reads and returns the value in M_i.

Pseudo-code for our implementation is presented in Algorithm 3. For two arrays of registers, v_1 and v_2, we use **writearray**(v_1, v_2) to denote copying the value of v_2 into v_1 one register at a time. **readarray**(v_1) denotes reading from v_1 one register at a time and returning the concatenation of the values that were read. These operations are not atomic operations. We will use the convention that 0 represents FALSE and 1 represents TRUE.

The step complexities of READ and WRITE are $\Theta(\ell/k + (\log n)/k)$ because reading and incrementing V takes $\Theta((\log n)/k)$ steps, copying a buffer takes $\Theta(\ell/k)$ steps, and copying an announcement array takes $\Theta((\log n)/k)$ steps. Since both operations have bounded step complexity, the implementation is wait-free.

A total of $6n$ buffers, a size n announcement array, $5n$ bits, and a modulo $4n$ counter are used, so the overall space complexity is $\Theta(n\ell/k + n(\log n)/k)$.

5.3 Notation

Here we define the notation that we will use throughout the proof.

If O is an operation, $C(O, c)$ refers to the configuration immediately after O completes line c of the code for that operation. If line c of O is a function call, then $C(O, c)$ is the

configuration immediately after the linearization point of the function call. If line c of O is a **writearray** operation then $C(O, c)$ indicates the configuration immediately after the completion of the entire operation.

READ$_i$ is a READ operation performed by process p_i. WRITE$_i$ is a WRITE operation that helps reader p_i. This means $i = (version \bmod n)$ on line 2 of a WRITE$_i$ operation.

5.4 Properties of Handshaking Bits

The following two properties about the handshaking bits, $H_i.r$ and $H_i.w$, follow immediately from properties proved by Attiya and Welch [2].

P1. Let C be a configuration such that $H_i.r \neq H_i.w$ and let R be the last READ$_i$ operation such that $C(R, 3) \to C$. Then there exists a WRITE$_i$ operation W such that $C(R, 2) \to C(W, 7) \to C$.

P2. Let C be a configuration such that $H_i.r = H_i.w$ and let R be the last READ$_i$ operation such that $C(R, 3) \to C$. Then there does not exist a WRITE$_i$ operation W such that $C(R, 3) \to C(W, 6) \to C(W, 7) \to C$.

In addition, the handshaking bits satisfy the following two properties.

P3. If $H_i.r \neq H_i.w$ at C and $H_i.r = H_i.w$ at a later configuration C', then there exists a READ$_i$ operation R such that $C \to C(R, 3) \to C'$.

P4 If $H_i.r = H_i.w$ at C and $H_i.r \neq H_i.w$ at a later configuration C', then there exists a WRITE$_i$ operation W such that $C \to C(W, 7) \to C'$.

Both properties have analogous proofs so we will only present the proof of **P3**.

Proof. Suppose, for contradiction, that **P3** is violated in some execution. Let C be the first configuration at which $H_i.r \neq H_i.w$ and there exists a later configuration C' at which $H_i.r = H_i.w$, but there is no READ$_i$ operation R such that $C \to C(R, 3) \to C'$. Consider the earliest such configuration C'. $H_i.r$ is not written to between C and C' because $H_i.r$ is only written to by line 3 of a READ$_i$ operation. Suppose, without loss of generality, that $H_i.r = 0$ and $H_i.w = 1$ at C. Then $H_i.r = H_i.w = 0$ at C' and there exists a WRITE$_i$ operation W that changes $H_i.w$ from 1 to 0 between C and C' by performing line 7. This means $H_i.r = 1$ at $C(W, 6)$, so $C(W, 6) \to C$. There must have been a READ$_i$ operation R that changed $H_i.r$ from 1 to 0 by performing line 3 between $C(W, 6)$ and C. Thus, $H_i.w = 0$ at $C(R, 2)$. Since $C \to C(W, 7)$ and $H_i.w = 1$ between $C(W, 6)$ and $C(W, 7)$, it follows that $C(R, 2) \to C(W, 6)$. At $C(R, 2)$, $H_i.r = 1$ and $H_i.w = 0$. At $C(W, 6)$, $H_i.r = 1$ and $H_i.w = 1$. Since $C(R, 2) \to C(W, 6) \to C(R, 3)$, there does not exist a READ$_i$ operation R' such that $C(R, 2) \to C(R', 3) \to C(W, 6)$. This contradicts the definition of C. Therefore **P3** holds. ◀

5.5 Proof of Correctness

Consider an arbitrary execution of READ and WRITE operations. Each WRITE operation W is linearized at $C(W, 9)$, which is immediately after it increments V.

Let R be a READ$_i$ operation by process p_i. If the check on line 8 of R returns FALSE, then R is linearized at $C(R, 4)$. We will prove that the value R returns is equal to the value of $G[V]$ at line 4 of R. If the check on line 8 of R returns TRUE, then R is linearized at $C(W, 12)$, where W is the last WRITE$_i$ operation such that $C(R, 2) \to C(W, 7) \to C(R, 7)$. In this case, we will prove that R returns the value written by W.

All local variables, in particular *version*, are initialized to 0. All global variables are also initialized to 0. This initial configuration can be viewed as the result of a complete WRITE(0) operation. Thus every READ operation is preceded by a complete WRITE operation.

Intuitively, the following lemma says that, if the test on line 8 of a READ_i operation R returns TRUE, then a WRITE_i operation has acknowledged the help request from R. This lemma also shows that the linearization point of R exists and is within its execution interval.

▶ **Lemma 2.** *Let R be a READ_i operation by process p_i. If $H_i.r \neq H_i.w$ at $C(R,7)$ and $F_{N_i} = 1$ at $C(R,8)$, then there exists a WRITE_i operation W such that $C(R,2) \to C(W,7) \to C(R,7)$. Furthermore, $C(W,12) \to C(R,8)$ for any such W.*

Proof. Suppose that $H_i.r \neq H_i.w$ at $C(R,7)$ and $F_{N_i} = 1$ at $C(R,8)$. By **P1**, there exists a write operation W such that $C(R,2) \to C(W,7) \to C(R,7)$. Let W be any such operation. Since line 4 of W sets F_{N_i} to 0 and $F_{N_i} = 1$ when p_i performs line 8 of R, F_{N_i} must have changed from 0 to 1 between $C(W,4)$ and $C(R,8)$. Therefore $C(W,12) \to C(R,8)$. ◀

Similarly, the following lemma says that if the test on line 13 of a READ_i operation R returns TRUE, then a WRITE_i operation has acknowledged the help request from R. Its proof can be obtained from the proof of the previous lemma by replacing $C(R,7)$ and $C(R,8)$ with $C(R,12)$ and $C(R,13)$.

▶ **Lemma 3.** *Let R be a READ_i operation by process p_i. If $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{N_i} = 1$ at $C(R,13)$, then there exists a WRITE_i operation W such that $C(R,2) \to C(W,7) \to C(R,12)$.*

Informally, the following lemma states that N_i will not change between any configuration where $H_i.r \neq H_i.w$ and $F_{N_i} = 1$, and the next execution of line 3 of a READ_i operation. This means that the reader can read safely from N_i on line 9.

▶ **Lemma 4.** *Let C be a configuration where $H_i.r \neq H_i.w$ and $F_{N_i} = 1$. Let R be the first READ_i operation such that $C \to C(R,3)$. Then N_i is not written to between C and $C(R,3)$.*

Proof. Suppose N_i was written to between C and $C(R,3)$. Then there must be a WRITE_i operation W such that some step of line 11 of W is executed between C and $C(R,3)$. From the code, $F_{N_i} = 0$ from $C(W,10)$ until W performs line 12. Since $F_{N_i} = 1$ at C, it follows that $C \to C(W,10)$ and F_{N_i} changed from 1 to 0 between C and $C(W,10)$. So line 4 of some write operation W' was performed in this interval. It follows from the code that $H_i.r = H_i.w$ at $C(W',3)$.

Suppose C occurred between $C(W',3)$ and $C(W',4)$. By **P4**, line 7 of some WRITE operation must have occurred between $C(W',3)$ and C, which is impossible since there is only one writer. Therefore C occurred before $C(W',3)$.

In this case, $H_i.r$ and $H_i.w$ changed from being unequal to being equal between C and $C(W',3)$. By **P3**, line 3 of some READ_i operation was performed between C and $C(W',3)$, which means it was performed between C and $C(R,3)$. This contradicts the choice of R. Therefore N_i cannot be written to between C and $C(R,3)$. ◀

The following lemma is an analogous statement for M_i. It guarantees that the reader can read safely from M_i on line 14. Its proof can be obtained from the proof of Lemma 4 by changing each occurrence of N_i to M_i and changing line numbers appropriately.

▶ **Lemma 5.** *Let C be a configuration where $H_i.r \neq H_i.w$ and $F_{M_i} = 1$. Let R be the first READ_i operation such that $C \to C(R,3)$. Then M_i is not written to between C and $C(R,3)$.*

For the remainder of this section, R will represent an arbitrary READ_i operation. We will show that R returns the input to the last WRITE operation linearized before R. Let v denote the value of V that process p_i reads on line 4 of R, and let g be the ℓ-bit value in $G[v]$ at configuration $C(R, 4)$. It follows from the next lemma that R must return g when R is linearized at $C(R, 4)$.

▶ **Lemma 6.** *g is the input to the last* WRITE *operation linearized before $C(R, 4)$.*

Proof. Let W be the last WRITE operation to be linearized before $C(R, 4)$. Suppose that g was not the input to W. Since W writes to $G[v]$ before it is linearized, another WRITE operation must have written to $G[v]$ between the linearization point of W and $C(R, 4)$. After W is linearized, V must be incremented $4n - 1$ times before *version* is reassigned value v on line 1 and $G[v]$ is written to on line 8. Thus, at least $4n - 1$ other WRITE operations are linearized after W and before $C(R, 4)$. This contradicts the choice of W. ◀

Lemma 7 is used to show that a complete WRITE_i operation between $C(R, 4)$ and $C(R, 7)$ will cause the check on line 8 of R to return TRUE. It is also used to prove Lemma 8.

▶ **Lemma 7.** *If there is at least one complete WRITE_i operation W between $C(R, 4)$ and the end of R, then $H_i.r \neq H_i.w$ and $F_{N_i} = 1$ from the end of W to the end of R.*

Proof. By lines 3–7 and 10–12 of the code for WRITE_i, it follows that $H_i.r \neq H_i.w$ and $F_{N_i} = 1$ at the end of W. No READ_i operation performs line 3 between $C(R, 4)$ and the end of R, so, by the contrapositive of **P3**, $H_i.r \neq H_i.w$ from the end of W until the end of R. This also implies that line 4 of the code for WRITE_i is not performed from the end of W until the end of R, so F_{N_i} remains equal to 1 until the end of R. ◀

Suppose there are two complete WRITE_i operations between $C(R, 4)$ and the end of R. Lemma 8 implies that no write to M_i occurs between the end of the second WRITE_i operation and the end of R. This will later be used to show that R reads g from M_i if it executes line 14.

▶ **Lemma 8.** *If the test on line 8 of R evaluates to* FALSE *and there are at least two complete WRITE_i operations between $C(R, 4)$ and the end of R, then $F_{M_i} = 1$ from the end of the second WRITE_i operation to the end of R.*

Proof. Let W and W' be the first and second WRITE_i operations between $C(R, 4)$ and the end of R. Suppose, for contradiction, that W finishes before $C(R, 7)$. Then, by Lemma 7, $H_i.r \neq H_i.w$ at $C(R, 7)$ and $F_{N_i} = 1$ at $C(R, 8)$, so the test on line 8 of R will evaluate to TRUE. This contradicts the assumption that line 8 of R evaluates to FALSE. Therefore W finishes after $C(R, 7)$ so W' starts after $C(R, 7)$.

From the code, we see that $F_{A_i} = 1$ from $C(R, 7)$ to the end of R, so $F_{A_i} = 1$ during the execution of W'. It follows from lines 13–16 of the code for WRITE_i that $F_{M_i} = 1$ at the end of W'. By Lemma 7, $H_i.r \neq H_i.w$ between the end of W and the end of R. Therefore line 5 of the code for WRITE_i is not performed from the end of W' until the end of R. Hence F_{M_i} remains equal to 1 from the end of W' until the end of R. ◀

Intuitively, the following lemma captures the idea that, as long as $H_i.r = H_i.w$ or $F_{M_i} = 0$, $G[v]$ is equal to g. This lemma shows that R must have read g on line 11 if the check on line 13 returns FALSE.

▶ **Lemma 9.** *If R read a value other than g from $G[v]$ on line 11, then $H_i.r \neq H_i.w$ at $C(R, 12)$ and $F_{M_i} = 1$ at $C(R, 13)$.*

Proof. Suppose that R read a value other than g from $G[v]$ on line 11. Note that, between $C(R,4)$ and $C(R,11)$, line 3 of READ_i is not performed, so the value of $H_i.r$ does not change.

By definition of g, a WRITE operation must have written to $G[v]$ between $C(R,4)$ and $C(R,11)$. Since V has value v at $C(R,4)$, V must be incremented $4n-1$ times after $C(R,4)$ before *version* is reassigned value v on line 1 of the WRITE and $G[v]$ is written to on line 8. This implies there are at least two full WRITE_i operations W' and W'' that helped process p_i between $C(R,4)$ and $C(R,11)$. Suppose W'' occurs after W'. By Lemma 7, $H_i.r \neq H_i.w$ and $F_{N_i} = 1$ from the end of W' until the end of R. Since R performed lines 11–13, the test on line 8 evaluated to FALSE. Therefore by Lemma 8, $F_{M_i} = 1$ from the end of W'' until the end of R. Since W'' finished before $C(R,11)$, it follows that $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{M_i} = 1$ at $C(R,13)$. ◀

The proof that R reads g on line 14 if the check on line 13 returns TRUE is presented below.

▶ **Lemma 10.** *If $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{M_i} = 1$ at $C(R,13)$, then R read g from M_i on line 14.*

Proof. Suppose $H_i.r \neq H_i.w$ at $C(R,12)$ and $F_{M_i} = 1$ at $C(R,13)$. By Lemma 3, there exists at least one write operation W such that $C(R,2) \to C(W,7) \to C(R,12)$. Let W be the last such write operation. Since W sets F_{M_i} to 0 on line 5 and $F_{M_i} = 1$ at $C(R,13)$, there exists a write operation W' (possibly equal to W) such that $C(W,5) \to C(W',16) \to C(R,13)$. Since there is only one writer, W' is either equal to W or starts after W finishes. Hence $C(W,7) \to C(W',13)$.

$F_{A_i} = 1$ at $C(W',13)$ because W' executed line 16. From the code, we see that $F_{A_i} = 0$ at $C(R,1)$ and only line 6 of a READ_i operation sets F_{A_i} to 1. Thus $C(R,6) \to C(W',13)$.

Next, we prove that *rversion* equals v at $C(W',14)$. Since $C(R,6) \to C(W',14) \to C(W',16) \to C(R,13)$, a complete execution of line 14 of W' occurs after R writes v to A_i on line 5 and before $C(R,13)$. From the code, we see that $A_i = v$ from $C(R,5)$ until the end of R, so $rversion = v$ at $C(W',14)$. Since *rversion* is a local variable, $rversion = v$ while W' performs line 15.

Suppose, for contradiction, that $G[v]$ was written to between $C(R,4)$ and $C(W',15)$. The writer writes to $G[v]$ before incrementing V to have value v on line 9. Since V has value v at $C(R,4)$, it follows that, between $C(R,4)$ and $C(W',15)$, V must have been incremented at least $4n-1$ times. Thus there are at least two complete WRITE_i operations between these two configurations. Since R performed lines 11–13, the test on line 8 evaluated to FALSE. By Lemma 8, $F_{M_i} = 1$ from the end of the second WRITE_i operation, W'', to the end of R. Since $C(R,4) \to C(W'',1) \to C(W',15) \to C(R,13)$, $F_{M_i} = 1$ at $C(W',15)$. This is impossible, since W' performs line 15 only if $F_{M_i} = 0$ at $C(W',13)$ and does not change the value of F_{M_i} until line 16. Therefore $G[v]$ was not written to between $C(R,4)$ and $C(W',15)$.

Be definition, $G[v] = g$ at $C(R,4)$, so $G[v] = g$ from $C(R,4)$ until $C(W',15)$. In particular, $G[v] = g$ while W' executed line 15. Hence $M_i = g$ at $C(W',15)$.

We claim that $H_i.r \neq H_i.w$ from $C(W,7)$ to the end of R. By the choice of W, we see that line 7 of a WRITE_i operation does not occur between $C(W,7)$ and $C(R,7)$. Since $H_i.r \neq H_i.w$ at $C(R,7)$, we know that $H_i.r \neq H_i.w$ from $C(W,7)$ to $C(R,7)$ by **P4**. From the code for READ_i, we see that line 3 of a READ_i operation is not executed between $C(R,7)$ and the end of R. Therefore by **P3**, $H_i.r \neq H_i.w$ from $C(R,7)$ to the end of R. Since $C(W,7) \to C(W',16) \to C(R,13)$, it follows that $H_i.r \neq H_i.w$ at $C(W',16)$.

From the code, we can see that $F_{M_i} = 1$ at $C(W',16)$ and M_i does not change between $C(W',15)$ and $C(W',16)$. Since $C(R,4) \to C(W',16)$, the first READ_i operation after

$C(W', 16)$ occurs after the end of R. So, by Lemma 5, M_i remains equal to g from $C(W', 16)$ to the end of R. In particular, $M_i = g$ at $C(R, 14)$. ◀

Finally, we show that R returns the input to the last WRITE operation linearized before R. First suppose that, $H_i.r \neq H_i.w$ at $C(R, 7)$ and $F_{N_i} = 1$ at $C(R, 8)$. By Lemma 2, there is a WRITE$_i$ operation W such that $C(R, 2) \to C(W, 7) \to C(R, 7)$. Let W be the last such WRITE$_i$ operation. Also by Lemma 2, line 12 of W is executed and $C(W, 12) \to C(R, 8)$. R is linearized at $C(W, 12)$ and W is linearized at $C(W, 9)$, so W is the last WRITE operation linearized before R. Let α be the input to W. Note that N_i equals α at $C(W, 12)$.

We claim that $H_i.r \neq H_i.w$ from $C(W, 7)$ to the end of R. By the choice of W, we see that line 7 of a WRITE$_i$ operation does not occur between $C(W, 7)$ and $C(R, 7)$. Since $H_i.r \neq H_i.w$ at $C(R, 7)$, we know that $H_i.r \neq H_i.w$ from $C(W, 7)$ to $C(R, 7)$ by **P4**. From the code for READ$_i$, we see that line 3 of a READ$_i$ operation is not executed between $C(R, 7)$ and the end of R. Therefore by **P3**, $H_i.r \neq H_i.w$ from $C(R, 7)$ to the end of R.

In particular, since $C(W, 7) \to C(W, 12) \to C(R, 8)$, $H_i.r \neq H_i.w$ at $C(W, 12)$. When W executes line 12, it sets F_{N_i} to 1. It follows from Lemma 4 that $N_i = \alpha$ from $C(W, 12)$ to line 3 of the next READ$_i$ operation after R, so N_i equals α at $C(R, 9)$. Thus, R returns the value written by W, as required.

Now suppose $H_i.r = H_i.w$ at $C(R, 7)$ or $F_{N_i} = 0$ at $C(R, 8)$. In this case, R is linearized at $C(R, 4)$. By Lemma 6, g is the value written by the last WRITE operation linearized before $C(R, 4)$. Thus, it suffices to prove that R returns g. Consider two subcases depending on the result of the test on line 13 of R. If the test returns false, then by the contrapositive of Lemma 9, the value that R read from $G[rversion]$ on line 11 is equal to g. If the test returns true, then by Lemma 10, R read g from M_i on line 14. So in either case, R returns g.

▶ **Theorem 11.** *Algorithm 3 implements an atomic ℓ-bit register.*

6 Step Lower Bound of Register Implementation

So far, we have presented implementations of an atomic ℓ-bit n-reader single-writer register from atomic k-bit n-reader single-writer registers. For the lower bound, we consider the case where the large register only needs to be regular and there is only one reader. This results in a stronger lower bound.

▶ **Theorem 12.** *Any regular ℓ-bit single-reader, single-writer register implementation from atomic k-bit single-writer registers with $O(\ell/k)$ step complexity for* READ *requires $\Omega(\ell/k)$ step complexity for* WRITE.

Proof. Let $t = \lfloor \ell/k \rfloor$. Since READ operations have step complexity $O(t)$, there are positive integers α and t_0 such that, for all $t \geq t_0$, each READ operation performs at most αt steps. Let $t \geq t_0$. We consider executions starting from an initial configuration I in which the writer completes its first WRITE, crashes, and then a single READ occurs. Let u be the number of steps performed by the writer in the worst case. If $u > \alpha t$, the lower bound holds, so suppose that $u \leq \alpha t$. The READ algorithm can be represented by a decision tree of height at most αt, where each node represents the read of a shared k-bit register. Without loss of generality, we may assume that no shared k-bit register is read more than once on any root to leaf path.

For each of the 2^ℓ different values w that can be written by the writer, consider the path in this decision tree taken by the reader. Let $E(w) = \{(i, v) \mid$ if the shared k-bit register read at depth i contains value $v \neq$ its value in $I\}$. The set $E(w)$ uniquely specifies the leaf that the reader reaches and, thus, is an encoding of w. Since $1 \leq i \leq u$ and there are $2^k - 1$

choices for each value v (i.e. excluding the value of the k-bit register in configuration I), the number of different possible encodings is

$$\sum_{j=0}^{u}\binom{\alpha t}{j}(2^k-1)^j \le \sum_{j=0}^{u}\binom{\alpha t}{u}\binom{u}{j}(2^k-1)^j = \binom{\alpha t}{u}\sum_{j=0}^{u}\binom{u}{j}(2^k-1)^j = \binom{\alpha t}{u}(2^k)^u$$
$$\le (2^k)^u(\alpha te/u)^u = (\alpha e\cdot 2^k t/u)^u.$$

Thus $2^\ell \le (\alpha e\cdot 2^k t/u)^u$. Taking the logarithm of both sides yields $\ell \le u(\log_2(\alpha e)+k+\log_2(t/u))$. Since $k\le \ell/t$ and $t<\ell$, it follows that $\ell \le u(\log_2(\alpha e)+\ell/t+\log_2(t/u))$ and $1\le (u/t)\cdot(\log_2(\alpha e)t/\ell+1+\log_2(t/u)t/\ell) < (u/t)\cdot(\log_2(\alpha e)+1+\log_2(t/u))$. Setting $\beta = t/u$ gives $1\le(\log_2(\alpha e)+1+\log_2\beta)/\beta$. Thus $\beta-\log_2\beta\le\log_2(\alpha e)+1\in O(1)$. This implies that $t/u=\beta\in O(1)$, because $\lim_{\beta\to\infty}\beta-\log_2\beta=\infty$. Therefore $u\in\Omega(t)=\Omega(\ell/k)$. ◀

7 Conclusion

We presented two new implementations of large ℓ-bit single-writer registers from small k-bit single-writer registers, which work for all $k\ge 1$. We can combine them as follows: use the first implementation if $\ell\le\lceil(\log_2 n)/2\rceil$; otherwise, use the second implementation. This results in an implementation of large single-writer registers with optimal step complexity and $\Theta(n\ell/k)$ space complexity.

We proved that any implementation with $O(\ell/k)$ step complexity for Read requires $\Omega(\ell/k)$ step complexity for Write. Since Read of an ℓ-bit register requires at least $\lceil\ell/k\rceil$ **reads** of k-bit registers, our lower bound shows that our implementation is step optimal.

It would be interesting to find a more space efficient implementation with optimal step complexity or to prove a lower bound on the amount of space required. We have some algorithms with $o(n\ell/k)$ space complexity, but slightly worse step complexity, so there may be a trade-off between space complexity and step complexity.

It would also be interesting to see if ℓ-bit multi-writer registers can be implemented from k-bit multi-writer registers with $\Theta(\ell/k)$ step complexity for any $k\ge 1$. Unfortunately, we do not know of any way to modify our tree based implementation to obtain a multi-writer regular (or atomic) register. Without an efficient counter that supports multiple incrementers, we can not extend our buffer based implementation to obtain a multi-writer register, either.

Acknowledgements. We would like to thank our supervisor, Professor Faith Ellen, for the many insightful discussions that we've had together, and for the numerous hours she put into editing our work.

References

1 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 385–395. ACM, 2014.

2 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

3 Soma Chaudhuri, Martha J. Kosa, and Jennifer L. Welch. One-write algorithms for multi-valued regular and atomic registers. *Acta Inf.*, 37(3):161–192, 2000.

4 Soma Chaudhuri and Jennifer L. Welch. Bounds on the costs of multivalued register implementations. *SIAM J. Comput.*, 23(2):335–354, 1994.

5 M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

6 A. Israeli and A. Shaham. Time and space optimal implementations of atomic multi-writer register. *Information and Computation*, 200(1):62–106, 2005.

7 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.

8 Andreas Larsson, Anders Gidenstam, Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Multiword atomic read/write registers on multiprocessor systems. *ACM Journal of Experimental Algorithmics*, 13, 2008.

9 G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

10 K Vidyasankar. A very simple construction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37(6):323–326, 1991.

Dynamic Atomic Snapshots*

Alexander Spiegelman[1] and Idit Keidar[2]

1 **Viterbi Dept. of Electrical Engineering, Technion, Haifa, Israel**
`sashas@tx.technion.ac.il`
2 **Viterbi Dept. of Electrical Engineering, Technion, Haifa, Israel**
`idish@ee.technion.ac.il`

Abstract

Snapshots are useful tools for monitoring big distributed and parallel systems. In this paper, we adapt the well-known atomic snapshot abstraction to dynamic models with an unbounded number of participating processes. Our *dynamic snapshot* specification extends the API to allow changing the set of processes whose values should be returned from a scan operation. We introduce the *ephemeral* memory model, which consists of a dynamically changing set of nodes; when a node is removed, its memory can be immediately reclaimed. In this model, we present an algorithm for wait-free dynamic atomic snapshots.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases snapshots, shared memory, dynamic, ephemeral memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.33

1 Introduction

Atomic snapshots [2, 12] are essential buildings blocks for distributed computing. For example, systems that perform long-running computations regularly take checkpoints in order to avoid restarting from scratch in case of failures [41, 40, 10, 35, 33, 19, 31]. Other systems use snapshots in order to gather statistics [9, 14] or to detect inconsistent states, (e.g., deadlocks) [32, 34, 17]. A snapshot API supports two operations: *scan* and *update*, where a *scan* returns a mapping from every participant to its last *update* value. Until now, snapshots were mostly considered in static models, where the set of participants cannot be dynamically changed.

Yet it is clear that long-lived reliable systems have to be able to replace old and faulty components with new ones. Indeed, there is a growing interest in *dynamic* distributed systems, in which the set of participating processes can be changed on-the-fly according to application demands and available resources [5, 27, 18, 38, 23, 15]. There is also strong motivation for checkpointing and monitoring dynamic systems, for example, large-scale distributed computations running on platforms like Hadoop [36] and Spark [20]. Another example is distributed block-chains [28, 11], which implement distributed shared memory, (e.g., a ledger); consistent snapshots of this memory can be useful for collecting statistical information and checking whether the system is subject to attacks [13].

Motivated by the above, we define and solve the *dynamic snapshot* problem. While previous work [16, 4] has addressed snapshots with infinitely many participants, (see Section 2), to the best of our knowledge, our snapshot is the first to allow dynamic changes in the set of participants whose values are returned by scan operations. We consider here asynchronous

* Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

dynamic shared memory consisting of *single-writer, multi-reader (SWMR)* registers, capturing systems in which every process has a private memory space where it publishes its state and all other processes can read from it; this occurs, for example, in map reduce-based computation platforms [36, 20], where each process stores partial computation results for later stages to process, as well as in state-machine replication [25, 21] and blockchain protocols [28], where one may want to monitor consistency across replicas.

We distinguish between *persistent* memory, where registers are available even after the processes that write to them are removed from the system, and *ephemeral* memory, which can be reclaimed. Once a process is removed, any ephemeral register it writes to can immediately become unavailable and thus be garbage collected. (Our model and problem definitions are given in Section 3.)

In order to implement any meaningful service in ephemeral memory we have to assume two essential conditions. First, a slow process may lose track of the active set of processes (ones that were added and not removed). Therefore, we have to equip the model with some *discovery* mechanism, which helps slow processes find new added ones. Second, the number of remove operations must be bounded. Otherwise, there is a scenario in which a slow process always tries to read from reclaimed memory, and is thus unable to complete operations; (see more details in Section 4).

Our main result is an atomic wait-free algorithm for dynamic snapshots in ephemeral memory. The algorithm is an extension of the well-known static snapshot algorithm by Attiya et al. [2]. The main challenges in making it dynamic are (i) tracking the active set of processes, (ii) dealing with a potentially infinite number of processes, which makes the helping mechanism more subtle, and (iii) making sure that no pertinent information is lost when ephemeral memory is reclaimed. For didactic reasons, we first present (in Section 5) an algorithm for the persistent dynamic memory model, overcoming the first two challenges. We then extend the algorithm (in Section 6) for ephemeral memory, addressing the third challenge. The complexity of every snapshot operation is quadratic in the number of processes that were added before the operation started, denoted m. An interesting question for future work is trying to reduce this complexity to $O(m \cdot log(m))$ as was done for static snapshots [8]; (see Section 7).

Summary of contributions

- We define the dynamic persistant and ephemeral memory models.
- We define a dynamic atomic snapshot.
- We implement wait-free dynamic atomic snapshots in both dynamic memory models.

2 Related Work

The atomic snapshot abstraction [2] was defined and widely studied in static systems, assuming a fixed set of participating processes. Shared memory models that allow infinitely many participating processes and snapshot implementations therein were previously presented in [16, 4]. As opposed to us, they assume multi-writer, multi-reader (MWMR) registers, which cannot be emulated from SWMR ones in these models (as proven in the full paper [39]). In addition, their implementations require a number of MWMR registers that is linear in the number of participating processes, and they do not allow memory reclamation. We, in contrast, define an ephemeral memory model in which registers pertaining to removed processes can be safely reclaimed.

The snapshot problem was also studied for concurrent data structures [24, 30, 29]. However, these works consider a different memory model than ours, in particular, all their memory objects are shared and are not "owned" by any of the threads. Thus, objects are not ephemeral in the sense of "disappearing" when their owners are removed. These papers more adequately capture shared memory multi-processors, whereas our model captures distributed systems with independent state per process.

Our dynamic shared memory models are inspired by recent dynamic work on dynamic message passing systems [5, 38, 23, 15], from which we adopt the idea that processes must be added via explicit *add* operations before they can invoke operations. Similarly, an explicit *remove* operation allows memory to be reclaimed. This extension allows us emulate snapshots from SWMR registers in the presence of infinitely many potential processes, which is impossible in shared memory models that do not support explicit add and remove [16, 4].

3 Model and Problem Definitions

We consider asynchronous dynamic memory, which extends asynchronous fault-prone memory [3, 22, 1] to allow for a dynamic set of nodes. We begin in Section 3.1 with standard shared memory definition, and continue in Section 3.2 to introduce dynamic memory. For brevity, some of the formal definitions can be found in the full paper [39] In Section 3.3, we define the *dynamic snapshot* abstraction, which we emulate in this paper.

3.1 Preliminaries

A shared memory model consists of an infinite set Π of processes accessing variables that reside at nodes from some set N.

Processes. Processes may *fail* by crashing or by invoking an explicit *stop* signal. A correct process is one that never fails. There is no restriction on the number of faulty processes.

Nodes. Each of the nodes is some shared memory location, either at a single server, or emulated by a group of servers that use protocols like ABD [6] and SMR [26] via message passing.

Processes access nodes' variables via *low-level operations* (e.g., read, write), and interact with objects emulated on top of the set of nodes via *high-level operations* (e.g., update and scan in a snapshot). Both high-level and low-level operations are *invoked* and subsequently *respond*. A *history* is a (finite or infinite) sequence of invocations and matching responses. We refer to the t^{th} event (invoke or response) in H as *time* t. An operation is *pending* in history H if its invocation occurs in H but its response does not.

Operation op_i *precedes* operation op_j in a history H, denoted $op_i \prec_H op_j$, if op_i's response occurs before op_j's invoke in H. Operations op_i and op_j are *concurrent* in H if neither precedes the other. A history with no concurrent operations is *sequential*. A history is *well-formed* if every process's subhistory is sequential. We consider only well-formed histories in this paper. We use sequential histories to define objects' correct behavior: an object's set of allowed sequential histories is called its *sequential specification*. The sequential specification of a register is the following: Every read operation returns the value of the last write that precedes it, or some initial value v_0 in case there is no such write.

Two histories of an object are *equivalent* if every process performs the same sequence of operations (with the same return values) in both, where operations that are pending in one can either be included in or excluded from the other. A *linearization* of a history H is an

equivalent sequential history that satisfies H's operation precedence relation and the object's sequential specification. An object is *atomic* if each of its histories has a linearization.

3.2 Dynamic memory

In the *dynamic* model, N is infinite, and the memory is actually kept at a finite subset of N, which changes dynamically. Objects in this model, called *dynamic* objects, have to provide a mechanism to reconfigure the system so as to change this subset. This is done via the special *add* and *remove* operations each object exposes. An explicit remove operation is essential for applications in order to be able to safely transfer a node's state before it is removed and becomes unavailable. An explicit add operation helps processes track the participating processes, as discussed in Section 4. Some initial subset $N_0 \subset N$ is known to all processes. We say by convention that for all $n \in N_0$, *add(n)* is invoked and responds at the beginning of every history. We say that a node n_i is *included* (respectively, *excluded*) at time t in history H if the prefix of length t of H includes a response of an $add(n_i)$ (respectively, $remove(n_i)$) operation. A node n_i is *active* in history H if it is included at any time in H and not excluded in H.

In this paper we are interested in what can and cannot be done assuming single writer registers. In this context, each node $n_i \in N$ is associated with a unique process $p_i \in \Pi$, and holds one atomic SWMR register to which only p_i can write and from which all processes can read. We refer to the SWMR register at node n_i, (which is associated with process p_i), as $segment_i$.

A process p_i is *active* if node n_i is active. A *wait-free* implementation of an object (in the dynamic model) is one that guarantees that any operation invoked by a correct (and active) process completes regardless of the actions of other processes.

We define two memory responsiveness models for dynamic memory:

- *Persistent memory:* Every $segment_i$ s.t. n_i is included is wait-free. That is, once a process is added, its segment is forever available.
- *Ephemeral memory:* Segments of active nodes are wait-free. Note that here, once a node is removed, the information it holds is not necessarily available.

Wait-free segments are called *responsive*, whereas other segments are *unresponsive* [22, 3, 1]. We refer to the dynamic model with persistent memory as the *persistent memory* model, and to the dynamic model with ephemeral memory as *ephemeral memory* model.

3.3 Dynamic snapshots

Snapshot objects [2] expose an interface for invoking *scan* and *update* operations. A *dynamic snapshot* object extends the snapshot object with *add* and *remove* operations, and has the following sequential specification:

▶ **Definition 1** (Dynamic snapshots' sequential specification)**.** Update, add, and remove return ok. A *scan* operation invoked at some time t in history H returns a mapping from every node n_i that is included and not excluded at time t in H to a value v_i s.t. v_i is the argument of the last *update* operation invoked by p_i before time t in H, or $\perp$ if no *update* is invoked by p_i before the *scan*.

In this paper we are interested in wait-free implementation of dynamic atomic snapshots in dynamic memory models.

4 Essential Assumptions

In this section we discuss our assumptions.

Explicit add. Wait-free high-level objects cannot be implemented from low-level SWMR registers if infinitely many processes may start to participate, i.e., (invoke high-level operations), at any time without an explicit add. This is actually true in both persistent and ephemeral memory models; (it is stated in [4], and, for completeness, proven in the full paper [39]. Thus, we henceforward assume the following:

▶ **Assumption 2.** *At any time, only processes associated with included nodes can invoke high-level operations.*

Discovery mechanism. Given that in ephemeral memory, removed nodes may be unresponsive, we have to equip processes with some mechanism to locate included nodes. Otherwise, a slow process may be unable to proceed after all nodes it had been aware of have been removed and have become unresponsive. For clarity, we avoid using an additional discovery entity, but instead assume that accesses to unresponsive nodes throw exception messages with segments belonging to responsive nodes. Formally, we assume the following:

▶ **Assumption 3.** *When a process p reads from an unresponsive node n_i, it receives either $segment_i$, or an exception notification with some $segment_j$. Moreover, if p reads n_i infinitely often and never receives $segment_i$, then every segment that belongs to a responsive node is returned at least once.*

Finite number of removals. In addition, it is impossible to implement wait-free dynamic objects in ephemeral memory in the presence of infinitely many remove operations. This can proven similarly to the impossibility proof in [37], and so the formal proof is omitted. Instead we provide the following intuitive justification:

▶ **Claim 4.** *There is no wait-free atomic snapshot implementation in ephemeral memory where infinitely many removes may be invoked.*

Proof Sketch. Consider a slow process p_i that invokes a high-level operation at time t and before its low-level operations reach any node, all nodes that were included by time t are removed and become unresponsive. We can construct an infinite history in which the following happens repeatedly: p_i learns from an exception about a node $n \in N$, then some other process p_j adds node $n' \neq n$ and removes node n. Notice that the add and remove operations have to be wait-free and p_j cannot write to the node associated with p_i (single writer), so the operations complete without affecting p_i's node. Then, node n becomes unresponsive, so p_i cannot read from it. By repeating this process infinitely, we get an infinite run where p_i does not read from any node except its own, and thus, its high-level operation cannot complete. A contradiction to wait-freedom. ◀

One way to circumvent the impossibility is by assuming a bound on the rate of remove operations and a corresponding bound on the low-level operation delay [7]. However, since we want to focus on a fully asynchronous model, we instead assume the following:

▶ **Assumption 5.** *The number of remove operations is finite.*

5 Dynamic Snapshots in Persistent Memory

In this section we assume Assumption 2 and present an algorithm for a wait-free dynamic snapshot in the persistent memory model. This algorithm serves as a stepping stone for our ephemeral memory algorithm given in the next section.

Static snapshots. The general idea is based on the well-known snapshot algorithm for static systems [2]: Each process p_i writes only to $segment_i$, which holds the value written by its last *update*, denoted val_i, and some additional information. A process that performs a *scan* operation repeatedly collects all the segments until it gets two identical scans, which is called a *double collect*. The process then stores in its segment the mapping of processes to data read from their segments in this double collect, called *view*. Notice that if other processes perform infinitely many updates concurrently with the *scan*, the scan may fail to ever obtain a double collect. In order to overcome this, the algorithm uses a helping mechanism, whereby a process obtains a scan and stores it in its view before writing a new value to its segment. A process that fails to obtain a successful double collect a certain number of times can "borrow" a view from another process.

Dynamic view. In the dynamic model, we need to implement also *add* and *remove*, which change the set of processes that can invoke operations and the set of values that should be returned by a *scan*. The view is thus no longer a static array. Instead, it is a mapping from a dynamic set of nodes to their values. Specifically, the view embedded in the segment holds three fields: The first, denoted *mem*, is the set of all known active nodes, initially N_0. (In the original algorithm, this set is static, thus there is no need to store it in the segment.) The second field, *removed*, tracks excluded nodes. The third field is a map, *snap*, from $mem \cup removed$ to segments, where $snap[i]$ holds the last value val_i read from $segment_i$.

In order for scans to determine which segments' values to return, (i.e., which nodes were included and not excluded), we add to every segment a set *changes* consisting of tuples of the form $\langle add/remove, n_i \rangle$. A process that performs *add* or *remove* adds the operation to *changes*. A scan by process p_i is performed in iterations as follows: It first collects the values from the segments that belong to processes in its current $mem \cup removed$; then checks their *changes* sets to discover which processes were included or excluded and updates *mem* and *removed* accordingly; and repeats this process if no double collect was obtained. Notice that since we consider a persistent memory model at this point, segments of excluded processes remain responsive. Therefore, information about added and removed processes is never lost, and even slow processes can obtain it.

Helping. The second issue we address is how a process can know which view it can borrow during a *scan* operation. Consider a run, illustrated in Figure 1, in which some process performs a *scan* concurrently with infinitely many *add* operations, s.t. every process performs exactly one *add* and no updates. One way for a scan to complete is by obtaining a successful double collect, but in this case, because of the infinitely many *add* operations, the *scan* can never obtain one despite the fact that there are no updates. Alternatively, a scan can borrow a view from another process, but it needs to make sure that the view is fresh enough.

To this end, we add a version number, denoted *num*, to every segment and include it in the embedded view. Each process increases its *num* at the beginning of every *scan* operation, and in every collect it checks whether some process has a view that contains its own updated

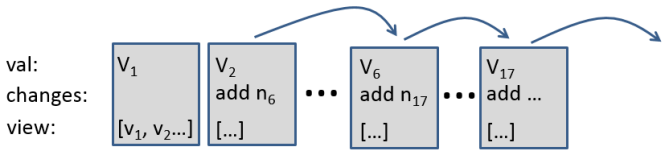


Figure 1 A run with infinitely many process additions; the scanning process cannot detect which view is fresh and may be borrowed.

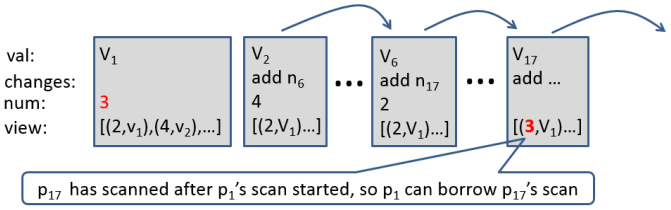


Figure 2 A run with infinitely many process additions; p_1's scan may return the view from $segment_{17}$, since it was obtained by p_{17} during the scan.

num. If some process has such a view, then it means that this view is fresh (obtained after the scan began) and can be borrowed. An illustration is presented in Figure 2.

Detailed algorithm. The segment structure is defined in Algorithm 1 and illustrated in Figure 3.

In the context of our algorithm, we say that a *node* n_i *is added before time* t if $n_i \in N_0$ or some process performs a low-level write of $\langle add, n_i \rangle$ to its segment's *changes* during an $add(n_i)$ operation before time t. In the same way, we say that a *node* n_i *is removed before time* t if some process performs a low-level write $\langle remove, n_i \rangle$ to its segment during $remove(n_i)$ before time t. These embedded writes are also the linearization points of the *add* and *remove* operations.

At any time t, we define *full-snapshot(t)* to be the states (excluding the embedded views) at time t of the segments of nodes added before time t: each node n_i is mapped to the tuple $\langle value, changes, num \rangle$ that was last written to $segment_i$ before time t. We define *snapshot(t)* to be the sub-mapping of *full-snapshot(t)* excluding nodes that were removed before time t.

The core of the algorithm lies in the *embeddedScan* procedure, which obtains *full-snapshot(t)* for some t that is later than the time when the procedure is invoked and saves it in the *view* field of the segment. Helping is done by performing *embeddedScan* at the beginning of every operation (*scan*, *update*, *add*, and *remove*).

Pseudocode for the algorithm's operations is presented in Algorithm 2. A *scan* first performs *embeddedScan* in line 2, and then in lines 3–5 it returns a mapping from scanned nodes in *mem* to their segment values. The *update* operation first performs *embeddedScan* and then writes the new value to the segment. Similarly, *add* and *remove* first perform *embeddedScan*, and then add to *changes* the information about the included or excluded node. Additionally, the initial value of a newly added segment is set as part of the *add* operation.

The *embeddedScan* procedure (Algorithm 3) first increases the version number (line 18), and then begins repeatedly collecting segments of all known processes. It uses two local variables to track the added nodes and their views, $CurView$ and $PrevView$. Each of them

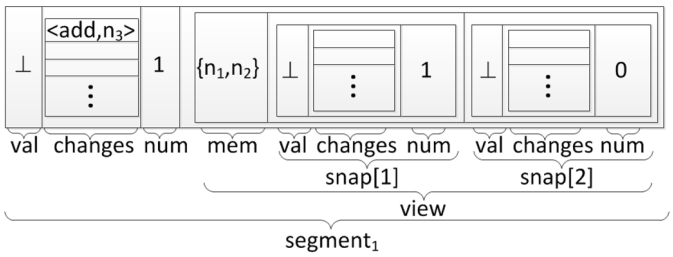


Figure 3 Example of $segment_1$. In this example $N_0 = \{n_1, n_2\}$, process p_2 has not invoked any operation yet, and process p_1 completed *add*(n_3), including writing 1 to $segment_1.num$, performing *embeddedScan* and writing the result to $segment_1.view$, and finally writing $\langle add, n_3\rangle$ to $segment_1.changes$.

Algorithm 1 Segment structure.

$segment = \langle val, changes, num, view\rangle$
  where $view = \langle mem, removed, snap\rangle$
    where $mem, removed \subseteq N$, and $snap$ is a mapping from mem to tuples $\langle val, changes, num\rangle$
initially: if $n_i \in N_0$, $segment_i = \langle \perp, \{\}, 0, \langle N_0, \langle \perp, \{\}, 0\rangle^{|N_0|}\rangle\rangle$, else $segment_i = \perp$

is structured like *view*, consisting of *mem*, *removed*, and *snap*. In every iteration after the first, $PrevView$ stores the view from the previous iteration, and in the first iteration it holds the view from p_i's segment. Lines 22-23 collect a new view into $CurView$. Note that we collect segments not only from nodes in the current *mem*, but also from removed ones. Failing to do so would introduce a subtle problem: it may cause us to miss operations that are successfully completed by processes after their removal, and before they discover the removal and stop; we shall revisit this issue in the next section, where we consider *ephemeral* memory and hence cannot rely on removed nodes to respond.

There are two ways for p_i to complete *embeddedScan*. The first is by obtaining a double collect in line 24. The second is by borrowing the view of another process that contains p_i's up-to-date version number (lines 27–29). It is guaranteed that this view was obtained after p_i's *embeddedScan* began because version numbers never decrease, and this number is increased at the beginning of the *embeddedScan*.

In lines 30–37, $PrevView$ is updated according to $CurView$. Finally, in line 38, p_i checks if its node was removed, and if so, stops. Otherwise, p_i writes the new view to its segment in line 39.

6 Dynamic Snapshots in Ephemeral Memory

In this section we assume Assumptions 2–5 and extend the algorithm of Section 5 for the ephemeral memory model. We present the algorithm in Section 6.1, and discuss its complexity in Section 6.2. A formal correctness proof is given in the full paper [39].

6.1 Algorithm

Recall that in the ephemeral memory model, nodes can become unresponsive, and thus, information (for example, about added and removed nodes) that is stored in their segments can be lost. Therefore, unlike the algorithm of Section 5, before removing a node, we need to make sure that information about its associated process's completed *add* and *remove* operations will persist after the node is excluded; note that it is possible that such operations

Algorithm 2 Dynamic snapshots in persistent memory: operations. Pseudocode for process p_i.

```
1:  procedure scan_i()
2:      embeddedScan()
3:      for each n_j ∈ segment_i.view.mem
4:          V[j] = segment_i.view.snap[j].val
5:      return V

6:  procedure update_i(d)
7:      embeddedScan()
8:      segment_i.val ← d

9:  procedure add_i(n_j)
10:     embeddedScan()
11:     segment_j ← ⟨⊥, {}, 0, segment_i.view⟩          ▷ set segment_j's initial value
12:     segment_i.changes ← segment_i.changes ∪ {⟨add, n_j⟩}

13: procedure remove_i(n_j)
14:     embeddedScan()
15:     segment_i.changes ← segment_i.changes ∪ {⟨remove, n_j⟩}
```

are still pending when the node is being removed and complete later. Our algorithm correctness is based on the following claim (see proof in the full paper [39]):

▶ **Claim 6.** *For every time t, for every two processes p_i, p_j, if $segment_j.changes$ includes $\langle remove, n_i, commit\rangle$ at time t, then at time t, $segment_j.changes$ includes every $\langle OP, NODE, commit\rangle$ ever included in $segment_i.changes$.*

Note, in particular, that Claim 6 implies that if p_i completes an operation after p_j removes it, that operation is already reflected in p_j. Given our assumption that the number of removes is finite (Assumption 5), Claim 6 implies that information about every succeeded operation is eventually stored at an active, and therefore responsive, node. Note that once this information is stored at some responsive node, then thanks to our discovery mechanism (Assumption 3), it is reachable by all correct processes. From this point, every correct process can eventually complete its *embeddedScan* as in the algorithm of the previous section.

State transfer. In order to make sure that information about added and removed nodes persists, processes now update their *changes* set with all such information observed during an *embeddedScan*. The new algorithm's *embeddedScan* procedure is presented in Algorithm 4. The segment structure remains as in Algorithm 1. The *embeddedScan* uses a local set $Changes$ to track the information observed during its iterations, and $segment.changes$ is updated according to $Changes$ at the end of the procedure.

When a process p tries to read from a removed node in line 9 during an *embeddedScan*, the discovery service may throw an exception with a value read from another segment. Upon such an exception (line 27), p checks whether the *removed* set in the view returned by the exception contains nodes that p did not know were removed. If so, p updates it local variables $PrevView$ and $Changes$, and jumps to the beginning of the next iteration (Loop) to collect from the new *mem* set. Otherwise, retries the read.

Algorithm 3 Dynamic snapshots in persistent memory: *embeddedScan* function. Pseudocode for process p_i.

```
16: procedure embeddedScan()_i
17:     PrevView ← segment_i.view
18:     segment_i.num ← segment_i.num + 1                    ▷ increase version number

19:     while true                                           ▷ try to obtain a consistent snapshot
20:         CurView.mem ← PrevView.mem
21:         CurView.removed ← PrevView.removed
22:         for each n_j ∈ CurView.mem ∪ CurView.removed      ▷ collect
23:             CurView.snap[j] ← ⟨segment_j.val, segment_j.changes, segment_j.num⟩
24:         if CurView=PrevView                              ▷ successful double collect
25:             goto Done
26:         for each n_j ∈ CurView.mem ∪ CurView.removed
27:             if segment_j.view.snap[i].num = segment_i.num    ▷ found a fresh snapshot
28:                 CurView ← segment_j.view
29:                 goto Done
30:             for each ⟨OP, n_l⟩ ∈ CurView.snap[j].changes \ PrevView.snap[j].changes
                                                             ▷ update view
31:                 if OP = add ∧ n_l ∉ PrevView.removed
32:                     PrevView.mem ← PrevView.mem ∪ {n_l}
33:                     PrevView.snap[l] ← ⟨⊥, {}, 0⟩
34:                 else
35:                     PrevView.mem ← PrevView.mem \ {n_l}
36:                     PrevView.removed ← PrevView.removed ∪ {n_l}
37:             PrevView.snap[j] ← CurView.snap[j]
    Done:
38:     if ∃j s.t. ⟨remove, n_i⟩ ∈ CurView.snap[j].changes then stop    ▷ n_i was excluded
39:     segment_i.view ← CurView
```

Additional phases in *add* and *remove*. Since removed nodes can be unresponsive, processes should not attempt to collect their segments during *embeddedScan*. However, this introduces a subtle problem: In the basic algorithm, a process can complete an *add* or *remove* operation long after it is removed. For example, it can complete an *embeddedScan*, then be removed by some other process, and then (without knowing that it has been removed) write to its *segment.changes*; recall that writing to *changes* is the linearization point of the operation. Since processes no longer collect removed segments, we cannot allow removed nodes to complete operations that might be missed by some future *embeddedScan*.

To overcome this problem, we use multiple phases in the *add* and *remove* operations. Pseudocode for the revised operations is given in Algorithm 5. At first, $add(n)$ calls *embeddedScan* and adds $\langle add, n, propose\rangle$ to its *changes* set (lines 34-36). The purpose of this phase is to announce ongoing operations, so that other processes can help complete them if necessary, while still being able to refrain from completing the *add* in case self-removal is observed. Tuples with propose are not taken into account when the sets *mem* and *removed* are updated during *embeddedScan* iterations (line 16). The second phase calls *embeddedScan* again (line 37). Recall that if *embeddedScan* observes its own removal has started by some process, it stops. Otherwise, the operation adds $\langle add, n, commit\rangle$ to its *changes* set (line 38).

A *remove* operation consists of three phases. A process p_i that performs $remove(n_j)$ first calls *embeddedScan*, then adds $\langle remove, n_j, prepare\rangle$ to its *changes* set. The purpose of this phase is to announce ongoing remove operations so that removed processes will

Algorithm 4 Dynamic snapshots in ephemeral memory: *embeddedScan* function. Pseudo-code for process p_i.

```
1:  procedure embeddedScan_i()
2:      PrevView ← segment_i.view
3:      Changes ← segment_i.changes
4:      segment_i.num ← segment_i.num + 1                    ▷ increase version number
5:      while true                                           ▷ try to obtain consistent snapshot
6:          CurView.mem ← PrevView.mem
7:          CurView.removed ← PrevView.removed
8:          for each n_j ∈ CurView.mem                       ▷ the following line may through an exception
9:              CurView.snap[j] ← ⟨segment_j.value, segment_j.changes, segment_j.num⟩
10:         if CurView=PrevView                              ▷ successful double collect
11:             goto Done
12:         for each n_j ∈ CurView.mem s.t. PrevView.snap[j] ≠ CurView.snap[j]
13:             if segment_j.view.snap[i].num = segment_i.num    ▷ found a fresh snapshot
14:                 CurView ← segment_j.view                 ▷ may through an exception
15:                 goto Done
16:             for each ⟨OP, n_l, commit⟩ ∈ CurView.snap[j].changes \ Changes   ▷ update view
17:                 if OP = add ∧ n_l ∉ PrevView.removed
18:                     PrevView.mem ← PrevView.mem ∪ {n_l}
19:                     PrevView.snap[l] ← ⟨⊥, {}, 0⟩
20:                 else
21:                     PrevView.mem ← PrevView.mem \ {n_l}
22:                     PrevView.removed ← PrevView.removed ⋃ {n_l}
23:             Changes ← Changes ∪ CurView.snap[j].changes
24:             PrevView.snap[j] ← CurView.snap[j]
    Done:                                                    ▷ no exceptions from here
25:     segment_i ← ⟨segment_i.value, Changes, segment_i.num, CurView⟩
26:     if ⟨remove, n_i, ∗⟩ ∈ segment_i.changes then stop

27: upon exception(Seg)
28:     if Seg.removed \ PrevView.removed ≠ {}               ▷ found new removed node, jump forward
29:         PrevView ← Seg.view
30:         Changes ← Seg.changes
31:         goto Loop
32:     else retry read
```

observe them and stop before committing new operations. In the second phase p_i calls *embeddedScan* again in order to check what operations p_j concurrently performs, i.e., what operations p_j has already proposed but has not yet committed, and then it proposes them together with its proposal by adding $\langle OP, NODE, propose\rangle$ to its *changes* set for every $\langle OP, NODE, propose\rangle$ it has observed in $segment_j.changes$ during its last *embeddedScan* together with $\langle remove, p_j, propose\rangle$. This phase enforces a "flag principle": if the removed node doesn't see its own remove and stop, then its proposal is seen and proposed together with the proposal to remove it. For example, if a process p_1 performs $add(n)$ or $remove(n)$ concurrently with a $remove(n_1)$ operation by another process p_2, then either (1) p_1 observes $\langle remove, n_1, prepare\rangle$ before committing its operation and stops, or (2) p_2 observes p_1's $\langle OP, n, propose\rangle$ and proposes it together with $remove(n_1)$.

In the third phase p_i calls *embeddedScan* again, but this time it serves two different purposes: First, as in *add*, it checks (at the end of the *embeddedScan*) if some other process already initiated removal, in which case it stops before committing its proposals. Second, it

checks if some other process has already committed a $remove(p_j)$, in which case it completes the operation without committing p_j's proposals. Otherwise, p_i commits all its proposals, i.e., it adds $\langle OP, NODE, commit\rangle$ to its *changes* set for every $\langle OP, NODE, propose\rangle$ it proposed in the second phase. The second check is essential because in case p_i observes that some other process p_k had removed p_j, it may be the case that p_k had missed some of p_i's proposals and committed p_i's removal without them. Hence, committing them know violates Claim 6.

The linearization point of an $add(n)$ or $remove(n)$ operation is when $\langle add, n, commit\rangle$ or $\langle remove, n, commit\rangle$ is added to a *changes* set of one of the segments for the first time (not necessarily by the process that invoked the operation).

6.2 Complexity

In this section we analyze the complexity of our algorithm. We measure complexity of an operation as the total number of memory accesses it performs, including ones that result in exceptions. Note that all the operations (*update*, *scan*, *add*, and *remove*) perform *embeddedScan* at most three times in addition to a constant number of low-level writes. Thus, the asymptotic complexity of all operations is equal to the complexity of the *embeddedScan* procedure. We assume that the discovery service does not return the same segment twice during the same while iteration (collect).

▶ **Claim 7.** *Let op be an embeddedScan invoked at time t by process p_i, and let m be the number of included nodes at time t. Then op's complexity is $O(m^2)$.*

Proof Sketch. We start by showing that *op* performs at most $O(m)$ collects. Note that after two iterations, *op* performs an additional collect only if there exists a $segment_j$ that is different in the current and in the previous collects, and $segment_j.view.snap[i].num < segment_i.num$. This can only happen if there is an operation by process p_j that is invoked before *op*, during which p_j writes to $segment_j$ after p_i reads $segment_j$ in the previous collect, and before p_i reads $segment_j$ in the current collect. By Assumption 2 and since we assume well-formed histories, the number of such operations is bounded by m. Thus, *op* performs $O(m)$ collects.

We now show that *op* successfully reads at most $O(m)$ segments in every collect. Assume in a way of contradiction that *op* reads more than $2m$ segments in some collect *col*. Therefore, *op* observes, before *col* begins, more than m nodes that were added after *op* was invoked. Thus, *op* observes in some segment $segment_j$, before *col* begins, at least one node whose addition was invoked after *op*. Therefore, *op* reads $segment_j.view.snap[i].num = segment_i.num$ before *col* begins, and thus completes without performing *col*. A contradiction.

By a similar argument and by the assumption that the discovery service does not return the same segment twice in the same collect, the number of exceptions *op* handles in every collect is $O(m)$. All in all, we conclude that the complexity of our algorithm is $O(m^2)$. ◀

In our analysis above m denotes the number of nodes that are included before *op* is invoked. However, we do not need to count in m excluded nodes that become unresponsive before *op* is invoked and the discovery service no longer returns them. Therefore, the complexity of the algorithm depends on the quality of the discovery service: the faster it is notified about excluded nodes, the less excluded nodes affect complexity. For example, if the discovery service is perfect and excluded nodes immediately become unresponsive, then the complexity of an *embeddedScan* does not depend on nodes that were excluded before it was invoked.

Algorithm 5 Dynamic snapshots in ephemeral memory: *add* and *remove* operations. The *update* and *scan* operations remain the same as in Algorithm 2. Pseudocode for process p_i.

33: **procedure** $add_i(n_j)$
34: $embeddedScan()$ ▷ phase 1: propose
35: $segment_j \leftarrow \langle \perp, segment_i.changes, 0, segment_i.view \rangle$ ▷ set $segment_j$'s initial value
36: $segment_i.changes \leftarrow segment_i.changes \cup \{\langle add, n_j, propose \rangle\}$
37: $embeddedScan()$ ▷ phase 2: commit
38: $segment_i.changes \leftarrow segment_i.changes \cup \{\langle add, n_j, commit \rangle\}$

39: **procedure** $remove_i(n_j)$
40: $embeddedScan()$ ▷ phase 1: prepare
41: $segment_i.changes \leftarrow segment_i.changes \cup \{\langle remove, n_j, prepare \rangle\}$
42: $embeddedScan()$ ▷ phase 2: propose
43: $ProposeSet = \{\langle *, *, propose \rangle \in segment_i.snap[j].changes\} \cup \{\langle remove, n_j, propose \rangle\}$
44: $segment_i.changes \leftarrow segment_i.changes \cup ProposeSet$
45: $embeddedScan()$ ▷ phase 3: commit
46: **if** $\langle remove, p_j, commits \rangle \notin segment_i.changes$
47: $CommitSet = \{\langle OP, NODE, commit \rangle \mid \langle OP, NODE, propose \rangle \in ProposedSet\}$
48: $segment_i.changes \leftarrow segment_i.changes \cup CommitSet$

7 Discussion

Atomic snapshots are essential building blocks in distributed systems. Clearly, any long-lived distributed system must support dynamism to replace old entities with new ones. In this paper, we addressed dynamic atomic snapshots for the first time. We defined asynchronous dynamic shared memory models consisting of a changing active set of nodes, each of which contains SWMR registers. We distinguished between the case in which nodes that are no longer part of the set can be reclaimed and become unresponsive (ephemeral memory), and the case in which nodes are always responsive (persistent memory). We then defined a dynamic snapshot object that allows users to change the set of processes whose values should be returned by a scan operation, and presented implementations of this object in the persistent and ephemeral memory models.

Our algorithm has quadratic time complexity, and since it is based on a quadratic-complexity static algorithm [2], we cannot expect any better from our algorithm. An interesting question for future research is to determine whether more efficient algorithms exist, given that for static snapshots, $O(m \cdot log(m))$ algorithms are known [8].

Our notion of ephemeral memory is interesting in its own right because of its generality. It can be applied to message-passing models: Each node can be emulated on top of a number of servers (e.g., using ABD [6]), and our responsiveness definition abstracts away the need to deal explicitly with the failure model of the emulation algorithm. Therefore, another interesting future direction is to try to implement dynamic reliable storage [5, 38, 23, 27] in the ephemeral memory model.

References

1 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993. doi:10.1145/153724.153741.

3 Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. Benign failure models for shared memory. In *Distributed Algorithms*. Springer, 1993.

4 Marcos K Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, 35(2):36–59, 2004.

5 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011. doi:10.1145/1944345.1944348.

6 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1), 1995.

7 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Simulating a shared register in an asynchronous system that never stops changing. In *International Symposium on Distributed Computing*, pages 75–91. Springer, 2015.

8 Hagit Attiya and Ophir Rachman. Atomic snapshots in $o(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.

9 Jim Basney and Miron Livny. Managing network resources in condor. In *hpdc*, pages 298–299, 2000.

10 Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *ACM SIGARCH Computer Architecture News*, 32(5):235–247, 2004.

11 Christian Cachin. Architecture of the hyperledger blockchain fabric, 2016.

12 K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

13 Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

14 Naser Ezzati-Jivan and Michel R. Dagenais. A framework to compute statistics of system parameters from very large trace files. *ACM SIGOPS Operating Systems Review*, 47(1):43–54, 2013.

15 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 140–153. Springer, 2015.

16 Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 161–169. ACM, 2001.

17 MD Gan, ZJ Ding, SG Wang, WH Wu, and MC Zhou. Deadlock control of multithreaded software based on petri nets: A brief review. In *2016 IEEE 13th International Conference on Networking, Sensing, and Control (ICNSC)*, pages 1–5. IEEE, 2016.

18 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.

19 Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini, and Kei Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9. IEEE Computer Society, 2005.

20 Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

21 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

22 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3), 1998.

23 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *Proceedings of the 29th International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

24 Nikolaos D. Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *Proc. of the 19th International Conference On Principles Of Distributed Systems (OPODIS'15)*, volume 46 of *LIPIcs – Leibniz International Proceedings in Informatics*, pages 27:1–27:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.OPODIS.2015.27.

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

26 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

27 Nancy Lynch and Alex A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing*, pages 173–190. Springer, 2002.

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

29 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 239–248. IEEE, 2015.

30 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Of concurrent data structures and iterations. In *Algorithms, Probability, Networks, and Games*, pages 358–369. Springer, 2015.

31 Rolf Riesen, Kurt Ferreira, Duma Da Silva, Pierre Lemarinier, Dorian Arnold, and Patrick G Bridges. Alleviating scalability issues of checkpointing protocols. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

32 Alexander Borisovitch Romanovsky and Yi-Min Wang. Method for deadlock recovery using consistent global checkpoints, September 2 1997. US Patent 5,664,088.

33 Jose Carlos Sancho, Fabrizio Petrini, Kei Davis, Roberto Gioiosa, and Song Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2005.

34 Peter Scheuermann and Hsiang-Lung Tung. A deadlock checkpointing scheme for multidatabase systems. In *Research Issues on Data Engineering, 1992: Transaction and Query Processing, Second International Workshop on*, pages 184–191. IEEE, 1992.

35 Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38. IEEE Computer Society, 2004.

36 Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

37 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. *arXiv preprint arXiv:1507.07086*, 2015.

38 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: A tutorial. In *OPODIS*, pages 259–259, 2015.

39 Alexender Spiegelman and Idit Keidar. Dynamic atomic snapshots. Technical Report CCIT 907, EE, Technion, November 2016. URL: `http://webee.technion.ac.il/publication-link/index/id/703`.

40 Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. *Pittsburgh Supercomputing Center, Tech. Rep*, 2001.

41 Zheng Zhang. Checkpoint computer system utilizing a fifo buffer to re-synchronize the memory systems on the detection of an error, May 8 2001. US Patent 6,230,282.

Deletion without Rebalancing in Non-Blocking Binary Search Trees*

Meng He[1] and Mengdu Li[2]

1 **Faculty of Computer Science, Dalhousie University, Halifax, Canada**
mhe@cs.dal.ca
2 **Dark Matter LLC, Mississauga, Canada**
meng.du.li@dal.ca

Abstract

We present a provably linearizable and lock-free relaxed AVL tree called the *non-blocking ravl tree*. At any time, the height of a non-blocking ravl tree is upper bounded by $\log_\phi(2m) + c$, where ϕ is the golden ratio, m is the total number of successful INSERT operations performed so far and c is the number of active concurrent processes that have inserted new keys and are still rebalancing the tree at this time. The most significant feature of the non-blocking ravl tree is that it does not rebalance itself after DELETE operations. Instead, it performs rebalancing only after INSERT operations. Thus, the non-blocking ravl tree is much simpler to implement than other self-balancing concurrent *binary search trees* (BSTs) which typically introduce a large number of rebalancing cases after DELETE operations, while still providing a provable non-trivial bound on its height. We further conduct experimental studies to compare our solution with other state-of-the-art concurrent BSTs using randomly generated data sequences under uniform distributions, and find that our solution achieves the best performance among concurrent self-balancing BSTs. As the keys in access sequences are likely to be partially sorted in system software, we also conduct experiments using data sequences with various degrees of presortedness to better simulate applications in practice. Our experimental results show that, when there are enough degrees of presortedness, our solution achieves the best performance among all the concurrent BSTs used in our studies, including those that perform self-balancing operations and those that do not, and thus is potentially the best candidate for many real-world applications.

1998 ACM Subject Classification E.1 [Data Structures] Distributed Data Structures

Keywords and phrases concurrent data structures, non-blocking data structures, lock-free data structures, self-balancing binary search trees, relaxed AVL trees

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.34

1 Introduction

Concurrent data structures play an important role in modern multi-core and multi-processor systems, and extensive research has been done to design data structures that support efficient concurrent operations. As BSTs are fundamental data structures, many researchers have studied the problem of designing concurrent BSTs. Lock-based BSTs are the most intuitive solutions and have been shown to be efficient [23, 1, 18, 3, 9, 7, 8]. There is, however, a potential issue: if a process holding a lock on an object is halted by the operating system, all the other processes requiring access to the same object will be prevented from making any

* This work was supported by NSERC. The work was done when the second author was in Dalhousie University.

further progress. Non-blocking (lock-free) BSTs have thus been proposed in recent years to overcome this limitation [11, 10, 5, 24, 6, 21, 19].

Despite the extensive work on lock-based and non-blocking BSTs, only a few solutions support self-balancing [3, 7, 8, 5, 9, 18]. Self-balancing BSTs are important in both theory and practice: In theory, they have better bounds on query and update time than BSTs that do not support self-balancing; for real-world applications, studies [22] have shown that self-balancing BSTs outperform BSTs without self-rebalancing greatly in sequential settings. However, the current status of the research on concurrent BSTs is that there is much less work that provides non-trivial bounds on access time than the research on the sequential counterparts. In addition, almost all existing empirical studies [1, 3, 5, 9, 16, 19, 21, 24, 23] are performed using randomly generated data under uniform distributions. This approach, however, has some drawbacks. As mentioned in [22, 5, 1], such experimental settings favor BSTs without self-balancing greatly: the expected heights of these trees are logarithmic with high probability under these settings [15], while they avoid the costs of rebalancing. Studies [22] have also shown that it is common that in real-world applications, the keys in an access sequence are partially sorted, i.e., have some degree of presortedness, instead of being completely random. As an example, when entering student grades of a course into a database, the data are likely to be entered in the order of student IDs or names. Thus, random data do not simulate these scenarios well. Therefore, much work is needed to provide more theoretical results on concurrent self-balancing BSTs, and better designed experimental studies are also needed to evaluate their performance.

While many researchers are designing concurrent BSTs, some significant progress has also been made recently in the study of self-balancing BSTs in sequential settings. In particular, Sen and Tarjan [26] proposed a solution to address the issue that self-balancing BSTs introduce so many cases when performing rebalancing after Delete operations that many developers resort to alternative solutions. These alternative solutions, however, may have inferior performance. To provide developers with a viable self-balancing BST solution for the fast development required in industry, Sen and Tarjan came up with a relaxed AVL tree called the *ravl tree*. A ravl tree only rebalances itself after Insert operations, while its height is still bounded by $O(\log m)$, where m is the number of Insert operations performed so far. The total number of rebalancing cases in the ravl tree is incredibly few, posing a great advantage in software development.

Based on the state of the art of the research on concurrent and sequential BSTs as described above, we study the problem of designing a non-blocking self-balancing BST that only rebalances itself after Insert operations, while still providing a non-trivial provable bound on its height in terms of the total number of successful Insert operations performed so far and the number of active concurrent processes. As in sequential settings, such a solution will decrease the development time greatly in practice. Furthermore, it may even potentially improve throughput in concurrent settings: If threads performing Delete operations do not rebalance the tree after removing items, they can terminate sooner so that there are fewer concurrent threads in the system.

1.1 Our Work

We design a concurrent self-balancing BST called non-blocking ravl tree that only rebalances itself after Insert operations for asynchronous systems where shared memory locations can be accessed by multiple processes. The number of rebalancing cases introduced is much fewer than other non-blocking self-balancing BSTs such as the non-blocking chromatic tree proposed by Brown et al. [5]. More precisely, the non-blocking ravl tree only has 5 rebalancing

cases, while 22 cases have to be considered for the non-blocking chromatic tree. We prove the linearizability and progress property of a non-blocking ravl tree, and bound its height. The theoretical results of our research are summarized in the following theorem:

▶ **Theorem 1.** *The non-blocking ravl tree is linearizable and lock-free, and it only rebalances itself after* Insert *operations. For a non-blocking ravl tree built via a sequence of arbitrarily intermixed* Insert *and* Delete *operations from an empty tree, at any time during the execution, the height of the tree is bounded by* $\log_\phi(2m) + c$*, where* $\phi = \frac{1+\sqrt{5}}{2}$ *is the golden ratio,* m *is the number of* Insert *operations that have successfully inserted new keys into the tree so far and* c *is the number of* Insert *operations that have inserted a new item but not yet terminated at this time.*

We further conduct experimental studies to evaluate the performance of our solution by comparing it against other state-of-the-art concurrent BSTs. We first use randomly generated data under uniform distributions, and show that non-blocking ravl trees outperform other concurrent self-balancing BSTs. We then use sequences with different degrees of presortedness. Experimental results show that our solution achieves the best performance among all concurrent BSTs with or without self-balancing when the data sequences have enough degree of presortedness, so that the average heights of BSTs without self-balancing are approximately 4-5 times greater than the average heights of self-balancing BSTs. Considering that previous studies [22] have shown that, when implemented in system software, it is very common that BSTs without self-balancing are more than five times taller than self-balancing BSTs, we believe that our solution is the best candidate for many real-world applications.

When designing the non-blocking ravl tree, our main strategy is to apply the general approach proposed by Brown et al. [5] for developing lock-free BSTs to design a concurrent version of Sen and Tarjan's ravl tree, and thus existing techniques are borrowed. Due to the special nature of the problem studied, we develop a number of new twists on these ideas: The three rebalancing cases of the original ravl tree are no longer sufficient to cover all possibilities in concurrent settings, and thus, we consider five rebalancing cases. Furthermore, a correctness proof is provided to show that our approach indeed covers all possibilities, while no similar proofs are needed in Sen and Tarjan's work where the correctness is obvious. To bound the tree height, we still use the exponential potential function approach in Sen and Tarjan's work which was originally proposed by Haeupler et al. [17], but we develop a more complex potential function and prove more properties of rebalancing to complete the analysis. It is interesting to see that the exponential potential function approach can still be made to work despite the more complex cases in concurrent settings.

We also would like to point out that in sequential settings, ravl trees are one type of *rank-balanced trees* [17], in which balancing information called ranks are assigned to tree nodes, and by maintaining different invariants called *rank rules*, different results can be achieved. Certain rank rules can yield the classic AVL trees and red-black trees, and creative rules can be invented to design data structures with new properties, including the ravl tree and the *weak AVL* tree [17] which uses fewer rotations than previous work while ensuring that its height is never worse than a red-black tree. Our work can provide a general guideline for those who wish to design and study the concurrent versions of this new class of trees.

2 Related Work

A number of lock-based and non-blocking BSTs have been proposed. In this section, we introduce some state-of-the-art concurrent BSTs. We say that a BST is *unbalanced* if it does

not support self-balancing, and call a BST *external* if values are only stored in its leaves, while an *internal* BST uses all nodes to store values.

Lock-based search trees. Bronson et al. [3] proposed a lock-based variant of AVL trees based on *hand-over-hand optimistic validation* adapted from *Software Transactional Memory* (STM). It reduces the complexity of deleting a node with two children by simply setting the value stored in this node to *NULL* without physically removing it from the data structure. This is the only case in which nodes that do not store values may be created, and thus they call their trees *partially external*. The speculation-friendly BST proposed by Crain et al. [7] is an internal tree in which updates are split into transactions that modify the abstraction state and those that restructure the tree implementation. Drachsler et al. [9] proposed a partially non-blocking internal BST supporting lock-free Get operations and lock-based update operations via *logic ordering*. This technique applies to both unbalanced BSTs and AVL trees. The CITRUS tree proposed by Arbel and Attiya [1] is an unbalanced internal BST offering wait-free Get operations and lock-based updates. It allows concurrent updates on BSTs based on *Read Copy Update* (RCU) synchronization and fine-grained locking. Ramachandran and Mittal [23] proposed the CASTLE tree, an unbalanced lock-based internal BST which locks edges instead of nodes to achieve higher concurrency.

Lock-free search trees. The first non-blocking BST that has been theoretically proven to be linearizable and non-blocking was proposed by Ellen et al. [11]. It is an unbalanced external BST which implements *Compare-And-Set* (CAS) to perform update operations. In their solution, processes performing update operations help each other so the entire system is guaranteed to make progress. Ellen et al. [10] also proposed a variant of [11] in which the amortized cost of an update operation, op, is $O(h(op) + \dot{c}(op))$, where $h(op)$ is the height of the tree when op is invoked, and $\dot{c}(op)$ is the maximum number of active processes during the execution of op. An unbalanced non-blocking internal BST was proposed by Howley and Jones [19], in which both processes performing Get and processes performing update operations help other processes. In the unbalanced non-blocking external BST proposed by Natarajan and Mittal [21], each process performing update operations operates based on marking edges. Ramachandran and Mittal [24] proposed an internal unbalanced lock-free BST by combining ideas from [19] and [21]. Brown et al. [5] proposed a general technique for developing lock-free BSTs using non-blocking primitive operations [4]. They not only presented a framework for researchers to design new non-blocking BSTs from existing sequential or lock-based BSTs, but also provided guidelines to prove the correctness and progress properties of the new solutions.

3 Preliminaries

In this section, we describe the previous results that are used in our solution. Non-blocking ravl trees use primitive operations *Load-Linked Extended* (LLX) and *Store-Condition Extended* (SCX) proposed by Brown et al. [4] to carry out update steps. LLX and SCX operations are performed on *Data-record*s consisting of mutable and immutable user-defined fields. Immutable fields of a Data-record cannot be further changed after initialization. The mutable fields of a Data-record cannot be further changed after it has been finalized. A successful LLX operation performed on a Data-record r reads r's mutable fields and returns a snapshot of the values of these fields. If an LLX operation on r is concurrent with any SCX operation that modifies r, it is allowed to return *fail*. An LLX operation performed on a finalized Data-

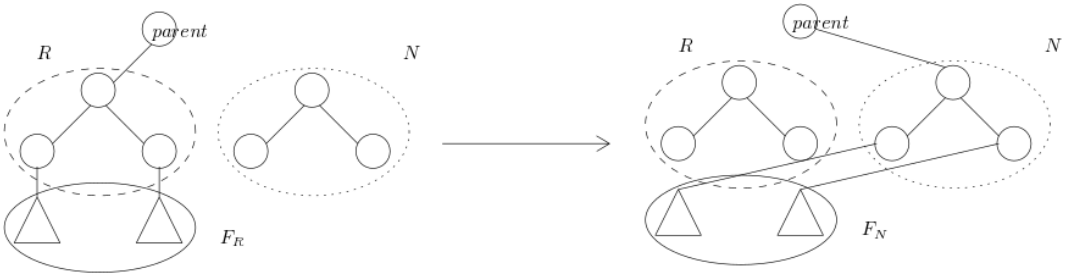


Figure 1 An example of a tree update operation following the template in [5].

record returns *finalized*. An SCX operation requires the following arguments: a sequence of Data-records V, a sequence of Data-records R which is a subset of V, a pointer *fld* pointing to a mutable field of a Data-record in V, and a new value *new*. A successful SCX operation atomically stores *new* into the mutable field pointed to by *fld* and finalizes all Data-records in R. An SCX operation can fail if it is concurrent with any other SCX operation performed by another process that modifies the Data-records in V.

Non-blocking ravl trees use the template proposed by Brown et al. [5] to perform updates. This template provides a framework to design a non-blocking *down tree*, which is a directed acyclic graph in which there is exactly one special root node with indegree 0 and all other nodes are of indegree 1. An update operation using this template atomically removes a subtree from the data structure and replaces it with a newly created subtree using LLX and SCX operations. More precisely, we define R to be the set of nodes in the removed subtree, N to be the set of nodes in the newly added subtree, $F_R = \{x|\text{parent of } x \in R \text{ and } x \notin R\}$ before the update, and $F_N = \{x|\text{parent of } x \in N \text{ and } x \notin N\}$ after the update. Then, the down tree G_R induced by nodes in $R \cup F_R$ before the update is replaced by the down tree G_N induced by nodes in $N \cup F_N$ after the update. Let *parent* be the parent of the root node of G_R in the down tree. Figure 1 gives an example.

An update operation *op* following the tree update template first performs a top-down traversal until it reaches *parent*. It then performs a sequence of LLX operation on *parent* and a contiguous set of its descendants related to the desired update. We define σ to be the set of nodes on which *op* performs these LLX operations, $F_\sigma = \{x|\text{parent of } x \in \sigma \text{ and } x \notin \sigma\}$ before the update, and the down tree G_σ to be the subgraph induced by $\sigma \cup F_\sigma$. If any of these LLX operations returns *fail* or *finalized*, *op* returns *fail*. Otherwise, *op* constructs the required arguments for an SCX operation, which are V, R, *fld* and *new*, and calls SCX to perform the desired update. Lemma 2 summarizes their results.

▶ **Lemma 2** ([5]). *Consider a down tree on which concurrent update operations are performed following the tree update template. Suppose that when constructing SCX arguments in this template, the following conditions are always met:* (1) *V is a subset of σ;* (2) *R is a subset of V;* (3) *fld points to a child pointer of parent, and parent is in V;* (4) *new is a pointer pointing to the root of G_N, and G_N is a non-empty down tree;* (5) *let old be the value of the child pointer pointed by fld before the update, then if old = NULL before the update operation, $R = \emptyset$ and $F_N = \emptyset$;* (6) *if old $\neq$ NULL and $R = \emptyset$, F_N only contains the node pointed to by old;* (7) *all nodes in N must be newly created;* (8) *if a set of concurrent update operations take place entirely during a period of time when no successful SCX operations are performed, the nodes in the sequence V constructed by each of these operations must be ordered in the same tree traversal order;* (9) *if $R \neq \emptyset$ and G_σ is a down tree, then G_R is a non-empty down tree whose root is pointed to by old, and $F_N = F_R$. Then, successful tree update operations*

are linearized at the linearization points of their SCX steps. If tree update operations are performed infinitely often, they succeed infinitely often, and are thus non-blocking.

4 Non-Blocking Ravl Trees

We now describe the non-blocking ravl tree, which is based on the sequential *ravl tree* [26]. We follow the definitions of V, *fld*, *old*, *new*, R, N, F_R and F_N in Section 3. Due to the page limit, the pseudocode and more details of some algorithms are omitted, and we only sketch our proof of Theorem 1. These omitted details be found in the second author's master's thesis [20].

4.1 The Structures and Algorithms of Non-Blocking Ravl Trees

Each node x in a non-blocking ravl tree is represented by a Data-record $< x.left, x.right, x.k, x.v, x.r >$, where $x.left$ and $x.right$ are pointers to x's left child and right child, respectively, $x.k$ is the key that identifies x, $x.v$ is the value stored in x, and $x.r$ is x's rank, which is used for rebalancing. $x.left$ and $x.right$ are mutable fields, and the other fields are immutable. If either one of x's children is missing, we conceptually add *missing* node(s) as its child/children. If both of x's children are missing, x is a *leaf*; otherwise, x is an *internal* node which is for routing purposes only, and we define $x.v = NULL$. If x is a missing node, $x.r = -1$; otherwise, $x.r$ is a non-negative integer. Let z be x's parent. x is called a *i-node* if $z.r - x.r = i$. x is an *i,j-node* if one of x's children is an i-node and the other is a j-node. If x is a 0-node, we call it a *violation*, and the edge (z, x) is called a *violating edge*. In this case, we also say that x is z's 0-*child*, and z is a 0-*parent*. No violation exists in external ravl trees after their rebalancing processes have terminated.

As mentioned previously, different rank rules can achieve different goals for rank-balanced trees. For example, in sequential settings, the original AVL tree is a rank-balanced binary tree in which each internal node is a 1,1-node or a 1,2-node and each leaf is a 1,1-node of rank 0. Our non-blocking ravl tree has the same rank rule as the sequential one: after rebalancing has been carried out, the rank of each parent is strictly larger than that of a child. A sequential ravl tree maintains such an invariant by promoting and demoting nodes and possibly rotating nodes after an insertion, where a promotion or a demotion respectively increases or decreases the rank of a node by 1. In concurrent settings, after an insertion is performed, we use the tree update template [5] to perform rotations and/or replacing some nodes by newly created nodes whose ranks differ by 1. During such a rebalancing process, however, more types of nodes may appear in concurrent settings. For example, a 0,0-node may be created as the result of a concurrent insertion into a non-blocking ravl tree, but it never appears at any time in the sequential case. Thus we carefully design new twists in the algorithms and their analysis in the rest of this section.

To avoid special cases, we introduce the *entry node*, which is the entry point of a non-blocking ravl tree. An empty tree contains an entry node with a single left child leaf, which are *sentinel nodes*. In a non-empty tree, the leftmost grandchild of the entry node is the actual root of the tree. The sentinel nodes in a non-empty tree are the entry node, its left child and its left child's right child. The keys, values and ranks of the sentinel nodes are ∞, $NULL$ and ∞, respectively. If a node is neither a missing node nor a sentinel node, we call it an *original node*. We define the height of a non-blocking ravl tree to be the height of the subtree consisting of all its original nodes.

We now describe the implementations of operations in non-blocking ravl trees. When reading the descriptions of these algorithms, readers can refer to Figure 2 for illustration.

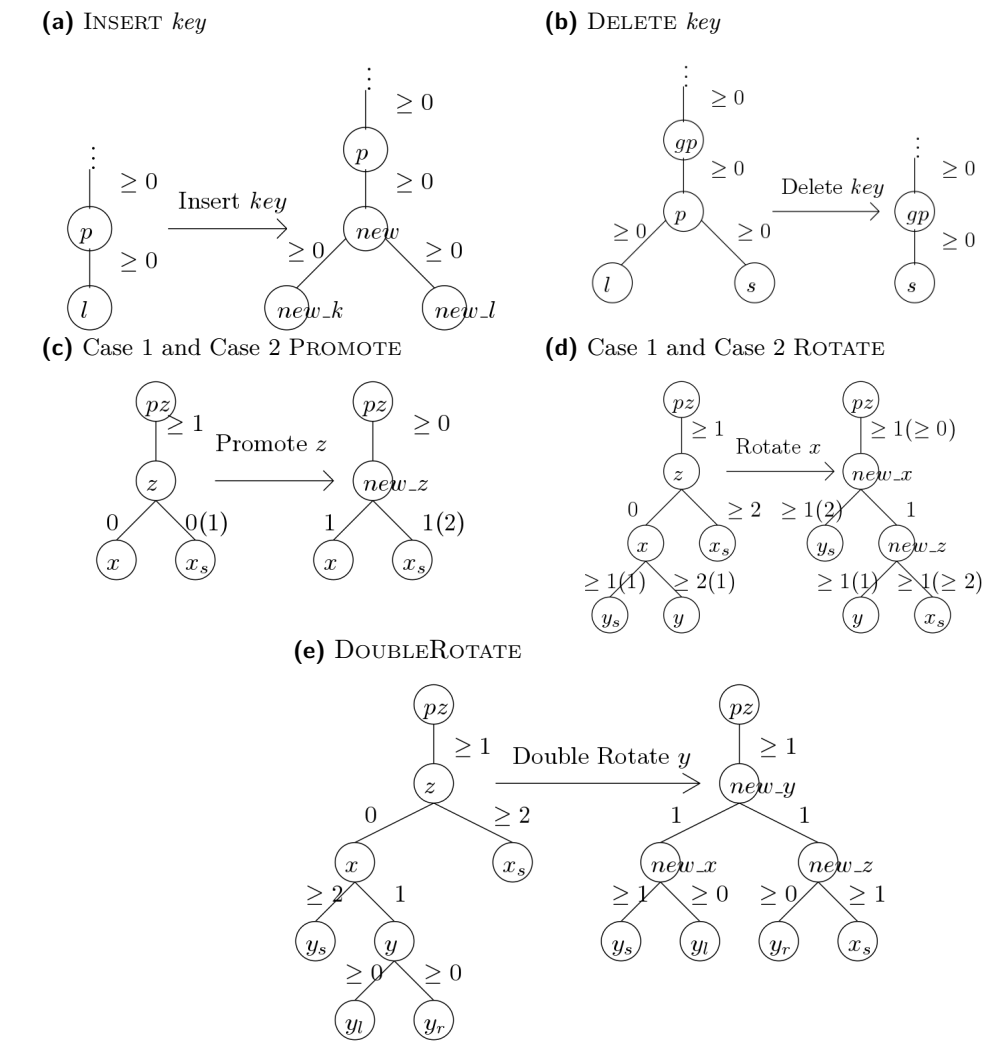


Figure 2 Operations in non-blocking ravl trees.

In this figure, only original nodes are shown. The number beside each edge shows the rank difference between the parent node and the child node.

A Search(key) operation performs a regular BST search starting from the entry node and returns the last three nodes visited, $< n_0, n_1, n_2 >$, where n_2 is a leaf, n_1 is n_2's parent, and n_0 is n_2's grandparent. A Get operation calls Search, and returns $n_2.v$ if $n_2.k = key$, or $NULL$ otherwise.

An Insert operation described in Algorithm 1 attempts to insert a new item consisting of key and $value$. This operation first calls Search(key), which returns a leaf l and its parent p in line 2. If $l.k$ is equal to key, then the key is already in the tree, and Insert returns $false$ without inserting any new item in line 3. Otherwise, it invokes TryInsert($p, l, key, value$) (Algorithm 2), which carries out the actual insertion shown in Figure 2(a) by following the tree update template summarized in Section 3. If TryInsert fails to insert this key, the Insert operation retries this process from scratch.

When performing the TryInsert operation, primitive operations such as LLX and SCX are called to implement the tree update template. Whenever these operations fail, or

Alg. 1 INSERT($key, value$)

```
1: repeat
2:    < −, p, l >← SEARCH(key)
3:    if l.k = key then return false
4:    result ← TRYINSERT(p, l, key, value)
5: until result ≠ fail
6: if result then CLEANUP(key)
7: return true
```

Alg. 2 TRYINSERT($p, l, key, value$)

```
8: if LLX(p) ∈ {fail, finalized} then
9:    return fail
10: if l = p.left then fld ← &(p.left)
11: else if l = p.right then
12:    fld ← &(p.right)
13: else return fail
14: if LLX(l) ∈ {fail, finalized} then
15:    return fail
16: new_k ← pointer to a new Data-Record
    < missing, missing, key, value, 0 >
17: if l = entry.left then
18:    new_l ← pointer to a new Data-
    Record < missing, missing, l.k, l.v, ∞ >
19: else
20:    new_l ← pointer to a new Data-
    Record < missing, missing, l.k, l.v, 0 >
21: if key < l.k then
22:    new ← pointer to a new Data-Record
    < new_k, new_l, l.k, NULL, l.r >
23: else
24:    new ← pointer to a new Data-Record
    < new_l, new_k, key, NULL, l.r >
25: if SCX({p, l}, {l}, fld, new)≠ fail then
26:    return (new.r = 0)
27: else return fail
```

Alg. 3 CLEANUP(key)

```
28: while true do
29:    gp ← NULL; p ← entry; l ←
    entry.left; l_s ← entry.right
30:    while true do
31:       if l is a leaf then return
32:       if key < l.k then
33:          gp ← p; p ← l; l ← l.left; l_s ←
    l.right
34:       else
35:          gp ← p; p ← l; l ← l.right; l_s ←
    l.left
36:       if p.r = l.r + 1 and p.r = l_s.r then
37:          TRYREBALANCE(gp, p, l_s)
38:          break out of the inner loop
39:       if p.r = l.r then
40:          TRYREBALANCE(gp, p, l)
41:          break out of the inner loop
```

whenever an LLX is performed on a Data-record that has been finalized, TRYINSERT returns immediately with failure. Otherwise, TRYINSERT successfully inserts a key, and returns a boolean value to indicate whether a violation has been created. More specifically, TRYINSERT first performs an LLX operation on p, and then, depending on if l is p's left child or right child, we let pointer *fld* point to the correct child pointer field of p (lines 8–12). If the structure of the related portion of the tree has been changed since the corresponding SEARCH operation returns, so that l is not a child of p anymore when we perform the check above, TRYINSERT returns *fail* in line 13. Then we perform an LLX operation on l in line 14, and create a new subtree rooted at the node pointed to by pointer *new* from line 16 to line 24: The key of the root of this new subtree is $\max(l.k, key)$ and its rank is set to $l.r$. This root has two children which are both leaf nodes, and they are pointed to by pointers *new_k* and *new_l*. The leaf node pointed to by *new_k* stores the key and value of the item to be inserted, and its rank is set to 0. The other leaf stores node l's key and value; its rank is set to ∞ if l was the left child of the entry node (i.e., *entry.left*) which is a sentinel node, and 0 otherwise. Next we construct the SCX arguments in line 25, where we define $V = \{p, l\}$ and $R = \{l\}$. The TRYINSERT operation then calls SCX with the constructed arguments and

attempts to atomically store *new* in the child field of p pointed to by *fld* while finalizing l. If this SCX operation fails, TryInsert returns *fail*. Otherwise, if the rank of the root of the newly inserted subtree is 0, TryInsert returns *true* to indicate that a new violation has been created after a successful insertion, and the corresponding Insert calls Cleanup (Algorithm 3) to rebalance the tree. If no new violation has been created, the TryInsert operation returns *false*. Finally, the corresponding Insert returns *true* to indicate that a new item has been inserted.

A Cleanup operation resolves the new violation created by the corresponding Insert operation, as well as all potential new violations created by the rebalancing steps during the process. Starting from the entry node (line 29), it performs a BST search for *key* and keeps track of the last three consecutive nodes visited, gp, p and l, as well as l's sibling, l_s. If p is a 0,1-node, and l_s is p's 0-child (line 36), the Cleanup operation calls TryRebalance(gp, p, l_s) to resolve the violation on l_s. This step is required to avoid livelocks. Otherwise, if l is a 0-node (line 39), the Cleanup operation calls TryRebalance(gp, p, l) to resolve the violation on l. Once the TryRebalance subroutine returns, the corresponding Cleanup operation retries this process in case that a new violation has been created by the previous TryRebalance call. The Cleanup operation returns when l reaches a leaf in line 31. At this point, the violation created by the corresponding Insert operation has been resolved.

A TryRebalance operation takes three consecutive nodes pz, z and x, which correspond to the nodes with same names in Figures 2(c)–(e). For Figure 2(c) and Figure 2(d), the numbers outside of the parenthesis show the rank difference for case 1 Promote and Rotate, respectively, while the numbers inside are for case 2 Promote and Rotate, respectively. Without loss of generality, assume that x is z's left child. Let y be x's right child, and y_s be x's left child. The TryRebalance operation attempts to resolve the violation on x following the tree update template. Based on the ranks of pz, z, z's sibling z_s, x and x's sibling x_s, the TryRebalance operation perform one of the following rebalancing steps: (1) if z is a 0,0-node or 0,1-node, it performs a case 1 or case 2 Promote on z as illustrated in Figure 2(c) by replacing z with a newly created node *new_z* whose rank is $z.r + 1$; (2) if z is a 0,i-node, where $i \geq 2$, there are three subcases: (2a) if $x.r \geq y.r + 2$ or y is a missing node, it performs a case 1 Rotate on x as illustrated in Figure 2(d) and replace x and z with newly created nodes *new_x* and *new_z* whose ranks are $x.r$ and $z.r - 1$, respectively; (2b) if $x.r = y.r + 1$ and $x.r = y_s.r + 1$, it performs a case 2 Rotate on x as illustrated in Figure 2(d) and replace x and z with newly created nodes *new_x* and *new_z* whose ranks are $x.r + 1$ and $z.r$, respectively; (2c) if $x.r = y.r + 1$ and $x.r \geq y_s.r + 2$, it performs a DoubleRotate operation on y as illustrated in Figure 2(e) and replace x, y and z with newly created nodes *new_x*, *new_y* and *new_z* whose ranks are $x.r - 1$, $y.r + 1$ and $z.r - 1$, respectively. If a rebalancing step does not introduce a new violation after resolving the old one, we say that it is *terminating*; otherwise, it is *non-terminating*.

A Delete operation follows the tree update template to remove a key as shown in Figure 2(b), and no rebalancing is needed after the tree is updated. It returns *false* if the key to be deleted does not exist in the tree, and *true* otherwise.

4.2 Properties of Non-Blocking Ravl Trees

In this section, we sketch our proof of Theorem 1. We first prove the correctness of our algorithms by arguing that the rebalancing operations described in Figures 2(c)–(e) cover all possible violation cases in concurrent settings. The only case that is not explicitly covered by our algorithm is when the 0-child of a 0,i-node, where $i \geq 2$, has a 0-child, and the following lemma shows that such a case does not occur.

▶ **Lemma 3.** *If a* 0*-node in a non-blocking ravl tree has at least one* 0*-child, then this node's parent is either a* 0,0*-node or a* 0,1*-node.*

We then prove that a non-blocking ravl tree remains a BST at any time, and thus all BST operations are performed correctly. For this, we define the *search path* [5] for a key k at any time to be the root-to-leaf path formed by the original nodes that a Search operation for k would visit as if this operation were performed instantaneously at this time. We then prove that a SEARCH operation in non-blocking ravl trees follows the correct search path at any time, and thus is performed correctly. We complete the proof by showing that the BST property of a non-blocking ravl tree is always preserved after update operations.

We next prove the linearizability of our solution. It would have been ideal to define the linearization point of a Search to be the time when it reaches a leaf node that is in the tree when visited. However, this claim cannot be guaranteed as a Search for a given key, k, in non-blocking ravl trees does not check the status of visited nodes, and by the time when it is linearized, this leaf might not be in the tree anymore. Thus, we prove that this leaf was in the tree and on the search path for k at some time earlier during this Search operation, so it is the correct node to return. We then define the linearization points of all operations:

▶ **Lemma 4.** *The linearization points of operations in non-blocking ravl trees are defined as follows:* (1) *a Search operation can be linearized when the leaf eventually reached was on the search path for the query key (such a time exists);* (2) *a Get operation is linearized at the linearization point of its Search step;* (3) *if an Insert operation returns true, it is linearized at the linearization point of the SCX step performed by its TryInsert step;* (4) *if an Insert operation returns false, it is linearized at the linearization point of its Search step;* (5) *if a Delete operation returns true, it is linearized at the linearization point of the SCX step by its TryDelete step;* (6) *if a Delete operation returns false, it is linearized at the linearization point of its Search step.*

Next, we show the progress properties of non-blocking ravl trees. We first show that if updates are invoked infinitely often, they follow the tree update template infinitely often. Thus, by Lemma 2, they will succeed infinitely often. We then construct a proof by contradiction to show that all operations in a non-blocking ravl tree are non-blocking. Consider a non-blocking ravl tree built via a sequence of arbitrarily intermixed Get, Insert and Delete operations. To derive a contradiction, assume that starting from a certain time T_1, active processes are still executing instructions, but none of them completes any operation. These active operations must include updates, because otherwise, the tree structure would remain stable and all Get operations would eventually terminate. We then use the property that update operations succeed infinitely often when called infinitely often and the fact that each update operation can succeed at most once, to argue that there exists a certain time T_2, after which the tree can only be modified by TryRebalance operations. We next observe that m Insert operations can only create at most $2m$ violations, and by carefully analyzing the relationship between different types of terminating and nonterminating rebalancing steps, we derive contradictions.

Finally, we bound the tree height. Consider a ravl tree T built via a sequence of arbitrarily intermixed Insert and Delete operations from an empty tree. At a time t, let m be the number of Insert operations that have successfully inserted new keys into T. Let T' be the balanced ravl tree rooted at rt' that can be constructed by completing all rebalancing operations to eliminate all the violations in T. We first bound the height of T' by augmenting the potential functions defined in [26] to include 0,0-nodes in our analysis. Here F_i denotes the ith Fibonacci number, i.e., $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ where $i \geq 2$. We also

use the inequality $F_{i+2} \geq \phi^i$. We define the potential of an original node x in an external ravl tree, whose rank is k, as follows: (1) if x is a 0,0-node, its potential is F_{k+3}; (2) if x is a 0,1-node, its potential is F_{k+2}; (3) if x is a 0,i-node, where $i \geq 2$, its potential is F_{k+1}; (4) if x is a 1,1-node, its potential is F_k; 5) otherwise, its potential is 0. We also define the potential of an external ravl tree to be the sum of the potentials of all its original nodes.

The keys steps of our analysis involves showing that each rebalancing step either does not change the potential of the tree, or decreases it by a certain amount. However, the analysis would not go through if a rebalancing step could also create a new 0,0-node. To ensure that this case never happens, we add additional checks in our rebalancing steps and prove the following lemma for non-terminating operations:

▶ **Lemma 5.** *A non-terminating operation that is either a* Promote *operation or a case* 2 Rotate *operation cannot create a new* 0,0*-node.*

From Figure 2(e), the node y on which a DoubleRotate is performed is allowed to have two 0-children. If this could indeed happen, our analysis would not work again. Thus we prove the next lemma which shows that y can have at most one 0-child in such a case:

▶ **Lemma 6.** *Consider a* DoubleRotate *operation as shown in Figure 2(e). The node y on which this operation is performed on can have at most one* 0*-child.*

We then perform case analysis to show that an Insert increase the potential of the tree by at most 2, while a Delete does not increase the tree potential. We also show that any non-terminating case 1 Promote/case 2 Promote/case 2 Rotate does not change the tree potential, while a case 1 Rotate or a DoubleRotate does not increase the tree potential. For a terminating case 1 Promote/case 2 Promote/case 2 Rotate, let k be the rank of the node that is promoted/rotated. Then such an operation decreases the potential of the tree by at most F_{k+2}. The potential decrease is exactly F_{k+2} if this terminating case 1 and 2 Promote operations is performed on the root, rt', of T', or if this case 2 Rotate operations is performed on one of rt''s children. Here we analyze the case 1 Promote as an example:

Let z be the 0,0-node on which a case 1 Promote operation is performed, and let k be its rank. The potential of z before the promotion was F_{k+3}. Let *new_z* be the newly added node that replaces z. Then, *new_z* is a 1,1-node, its rank is $k+1$, and its potential is F_{k+1}. Thus, replacing z by *new_z* decreases the potential of the tree by F_{k+2}. Let $p(z)$ be z's parent before the promotion. In the non-terminating case, by Lemma 5, $p(z)$ could not be a 1,0-node before this operation. If $p(z)$ was a 1,1-node, whose rank was $k+1$ before the promotion, it becomes a 0,1-node after the promotion, and its potential changes from F_{k+1} to F_{k+3}. If $p(z)$ was a 1,i-node, where $i \geq 2$, whose rank was $k+1$ before the promotion, it becomes a 0,i-node after the promotion, and its potential changes from 0 to F_{k+2}. In either case, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not changed. In the terminating case, if $p(z)$ was a 1,2-node, whose rank was $k+2$ before the promotion, it becomes a 1,1-node after the promotion, and its potential changes from 0 to F_{k+2}. If $p(z)$ was a 0,2-node with rank $k+2$ before the promotion, where z was its 2-child, $p(z)$ becomes a 0,1-node after the promotion, and its potential changes from F_{k+3} to F_{k+4}. In either case, the potential of $p(z)$ is increased by F_{k+2}, and the potential of the tree is not changed if z is not the root of the tree, i.e., $p(z)$ is an original node. If $p(z)$ was not a 1,2-node or 0,2-node, this promotion does not change its potential, and the tree potential is decreased by F_{k+2}. If z is the root of the tree, then $p(z)$ is a sentinel node. Since the potential of the tree is the sum of the potentials of its original nodes only, the tree potential is decreased by F_{k+2}.

We now make use of the above claims to bound the tree height. Initially, the potential of an empty tree is 0. Based on the analysis above, the potential of the tree can only be

increased by at most 2 after each Insert operation. The first Insert operation, where we insert the root of T', rt', into the empty tree, does not change the potential of the tree. Thus, after m Insert operations, the potential of the tree is at most $2(m-1)$. Since the initial rank of rt' is 0, the number of operations that increases the rank of rt' by 1, i.e., terminating case 1 and 2 Promote operations performed on rt' plus the number of case 2 Rotate operations performed on one of rt''s children, is equal to $rt'.r$. As analyzed above, each time one of these operations changes the rank of rt' from k to $k+1$, the potential of the tree is decreased by F_{k+2}. Therefore, these operations decrease the potential of the tree by $\sum_{i=0}^{rt'.r-1} F_{i+2} = \sum_{i=2}^{rt'.r+1} F_i = F_{rt'.r+3} - 2$. Since the potential of the tree is always non-negative, $2(m-1) \geq F_{rt'.r+3} - 2$, and $2m \geq F_{rt'.r+3} > F_{rt'.r+2} \geq \phi^{rt'.r}$. Therefore, $rt'.r < \log_\phi 2m$. We then show that the height of T' is no greater than $rt'.r$, which is bounded by $\log_\phi(2m)$ from the analysis above. Next, we borrow ideas from [5] to prove that the number of violating edges on each search path in T is bounded by the number, c, of active Insert operations that are in the rebalancing phase. Using these two claims, we bound the height of T by $\log_\phi(2m) + c$.

5 Experimental Evaluation

We compared the non-blocking ravl tree, **lf-ravl**, against the following concurrent BSTs: (1) **lf-chrm**, the non-blocking chromatic tree proposed by Brown et al. [5] which uses the tree update template summarized in Section 3; (2) **lf-chrm6**, a variant of lf-chrm in which the rebalancing process is only invoked by an update operation if the number of violations on the corresponding search path exceeds 6 [5]. Compared to lf-chrm, this variant achieves superior performance since it reduces the total number of rebalancing steps; (3) **lc-davl**, the concurrent AVL tree proposed by Drachsler et al. [9] which supports wait-free Get and lock-based updates; (4) **lf-nbst**, the unbalanced external non-blocking BST proposed by Natarajan and Mittal [21]; (5) **lf-ebst**, the unbalanced external non-blocking BST proposed by Ellen et al. [11]; (6) **lf-ibst**, the unbalanced internal non-blocking BST proposed by Ramachandran and Mittal [24]; (7) **lc-cast**, the unbalanced internal lock-based BST proposed by Ramachandran and Mittal [23]; (8) **lc-citr**, the unbalanced internal lock-based BST proposed by Arbel and Attiya [1]. We also implemented a variant of the non-blocking ravl tree called **lf-ravl3** in which Cleanup is only invoked by Insert if the number of violations on the search path exceeds 3. We allow at most three violations on a search path since experiments showed that this is when the tree achieved the best performance in most experimental settings.

The original source code for lf-chrm [5], lf-ebst [11] and lc-davl [9] was written in Java, and we re-implemented them in C. We used the source code for other concurrent BSTs developed by their original authors [21, 24, 23], and followed the framework in [16] to test the performance. CAS operations are performed using the APIs provided by `libatomic_ops` [2]. We also used `jemalloc` [14] for memory allocations to achieve optimal results. For lock-based data structures, we used mutex locks and the APIs provided by `pthread`. All experiments were conducted on a computer with two Intel® Xeon® E5-2650 v2 processors (20M Cache, 2.60 GHz) supporting 32 hardware threads in total. It operates on CentOS 6.7. All implementations were compiled using `gcc-4.4.7` with `-O3` optimization.

We first conduct experiments using random numbers as operation keys. All keys are positive integers within a user-specified range generated under uniform distributions. We also used different operation mixes. An operation mix xr-yi-zd represents x% Get operations, y% Insert operations and z% Delete operations. As in [9], we used operation mixes 90r-9i-1d

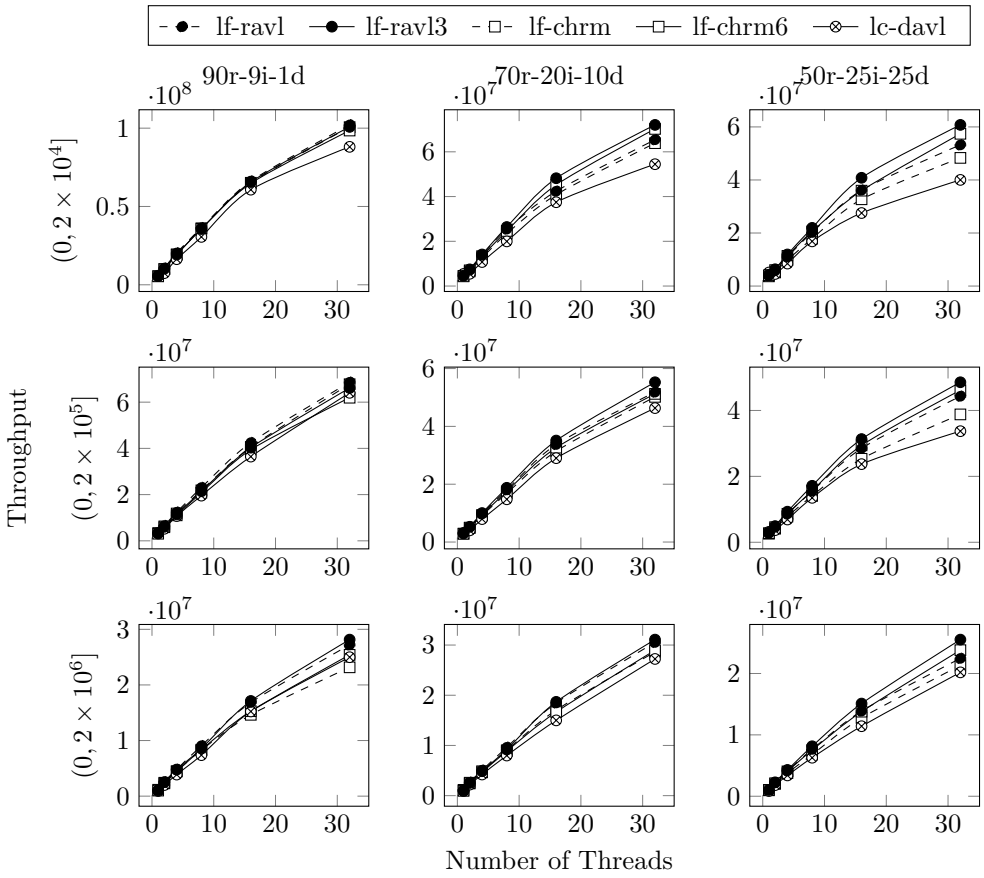


Figure 3 Experimental results comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs using randomly generated sequences under uniform distributions.

(read-dominant), 70r-20i-10d (mixed) and 50r-25i-25d (write-dominant). For each concurrent BST, under each key range and using each operation mix, we ran its program with 1, 2, 4, 8, 16 and 32 threads for 5 seconds as one trial. The performance of the data structures is measured using their average throughput (the number of operations finished per second). We prefilled each data structure using randomly generated keys until 50% of the keys in the key range are inserted into the tree.

Figure 3 shows the experimental results comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs. To test how different levels of contention can influence the performance, we test the algorithms on the following key ranges: $(0, 2 \times 10^4]$, $(0, 2 \times 10^5]$, and $(0, 2 \times 10^6]$. We group studies under the same key range by row, and studies using the same operation mix by column. The x-axis of each study shows the number of threads, and the y-axis shows the average throughput. For example, the studies in the first row shows results for studies under key range $(0, 2 \times 10^4]$. For this key range, lf-ravl outperforms lf-ravl3 slightly in case 90r-9i-1d. Since lf-ravl3 has a larger average search path length than lf-ravl, GET operations in lf-ravl are faster than in lf-ravl3. In other cases, lf-ravl3 achieves superior performance. This is because lf-ravl3 reduces the total number of rebalancing steps and introduces less overhead. lf-ravl outperforms lf-chrm in every case. Though lf-chrm6 outperforms lf-ravl in case 50r-25i-25d, lf-ravl3 still outperforms lf-chrm6. This shows that lf-ravl and lf-ravl3 improve performance by avoiding rebalancing after DELETE operations.

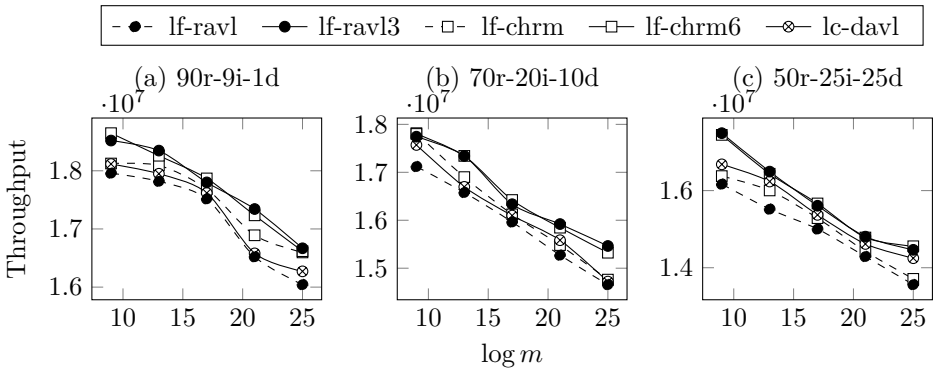


Figure 4 Experimental results comparing lf-ravl and lf-ravl3 against other self-balancing concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads).

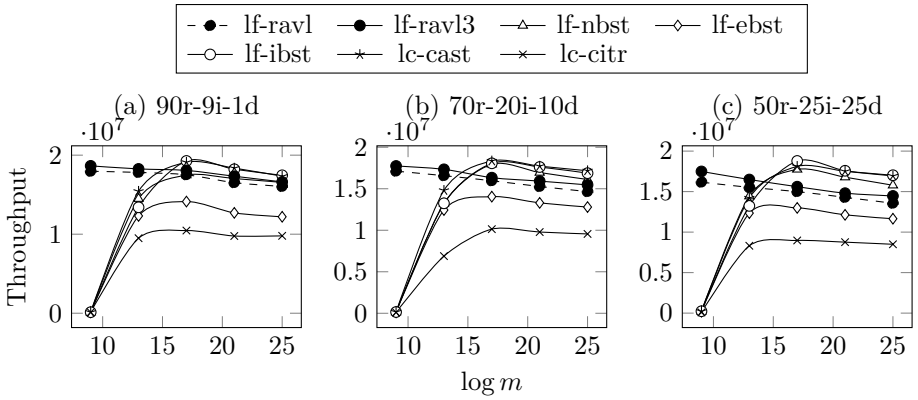


Figure 5 Experimental results comparing lf-ravl and lf-ravl3 against unbalanced concurrent BSTs using sequences of size 2^{26} with different degrees of presortedness (32 threads).

lf-ravl3 and lf-chrm6 both outperform lc-davl in every case. Results are similar for other key ranges, showing that lf-ravl and lf-ravl3 outperform other data structures in most cases. This performance difference is more noticeable with smaller key ranges (higher contention levels), which shows that our solution is more contention-friendly compared to other concurrent self-balancing BSTs.

Data generated from uniform distributions favor unbalanced BSTs, as they are balanced with high probability and do not have the overheads of rebalancing. Nevertheless, lf-ravl and lf-ravl3 still outperform some of the unbalanced BSTs in our studies. For example, they scale much better for operation sequences with a higher update ratio (70r-20i-10d and 50r-25i-25d) compared to lc-citr. The detailed experimental results are reported in [20].

We further test the performance using synthetic sequences in which keys are partially sorted, i.e., with a certain degree of presortedness. Presortedness has been extensively studied in adaptive sorting algorithms [12, 25, 13], and we apply this concept to the context of studying concurrent binary search trees. The presortedness of a sequence is measured by the number of *inversions*, which is the number of pairs of elements of the sequence in which the first item is larger than the second. We generate the inversions in a data sequence using the algorithm in [13, 25], which is expected to generate $mn/2$ inversions on average, where n is the size of the data sequence and m is a user-specified parameter to control the degree

of presortedness. We constructed data sequences with presortedness in the following way: starting from a sorted sequence of distinct positive integers of size $n = 2^{26}$, we generated sequences with different levels of presortedness by creating inversions using the algorithm in [12, 25, 13] with the values of m in range $[2^9, 2^{25}]$. For each data structure, using each data sequence and under each operation mix, we ran the program with 32 threads (which is the maximum number of hardware threads available in our setup) as one trial. Before each trial, we prefilled the data structure using the first half of each data sequence to ensure stable performance. During each trial, we insert items in the order of the data sequences. We also randomly selected keys within range $(0, 2^{26}]$ to perform GET and DELETE operations. Each trial terminated after all numbers in the data sequence had been inserted.

Figure 4 and Figure 5 give the experimental results comparing lf-ravl and lf-ravl3 against concurrent self-balancing BSTs and unbalanced BSTs, respectively, in which the x-axis shows the value of $\log m$ and the y-axis shows the throughput. lf-ravl3 outperforms other self-balancing BSTs in most cases. lf-chrm outperforms lf-ravl slightly. We believe that this is because the expected success rate of INSERT operations in the current experimental settings (100%) is higher than the previous studies (at most 50%), as lf-ravl rebalances the tree more often. lf-ravl3 outperforms lc-davl, while lf-davl achieves slightly better performance compared to lf-ravl.

We also consider how the value of m affect the heights of the BSTs when comparing the performance of our solution against other unbalanced concurrent BSTs. The average heights of our solution are not affected by the value of m significantly; the tree height is always between 30 and 34. When m is no greater than 2^9, the heights of unbalanced BSTs are notably larger than the heights of self-balancing trees, and lf-ravl and lf-ravl3 outperform unbalanced BSTs significantly. The average tree heights of unbalanced BSTs decrease from 41445 to 177 when m changes from 2^9 to 2^{15}, which explains their rapid performance improvement. When m is approximately 2^{15}, where unbalanced BSTs are approximately 5 times as tall as our solution, unbalanced BSTs start to have better performance. When m is larger than 2^{17}, unbalanced BSTs outperform our solution. Previous studies [22] have shown that, in real-world applications, it is very common for data to be accessed in some sorted order, and unbalanced BSTs are likely to be more than five times taller than self-balancing BSTs when implemented in system software products. In addition, if the comparisons between keys require more time (e.g., the keys are strings), the smaller heights of self-balancing BSTs will potentially be even more attractive. From the results and analysis above, we believe that our solution is the best candidate for many real-world applications.

References

1 Maya Arbel and Hagit Attiya. Concurrent updates with RCU: Search tree as an example. In *Proc. PODC*, pages 196–205, New York, NY, USA, 2014. ACM. doi:10.1145/2611462.2611471.

2 Hans Boehm. The atomic_ops library (libatomic_ops). URL: https://github.com/ivmai/libatomic_ops.

3 Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, January 2010. doi:10.1145/1837853.1693488.

4 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proc. PODC*, pages 13–22, New York, NY, USA, 2013. ACM. doi:10.1145/2484239.2484273.

5 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. *SIGPLAN Not.*, 49(8):329–342, February 2014. Source code available at `http://www.cs.toronto.edu/~tabrown/chromatic/`. `doi:10.1145/2692916.2555267`.

6 Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proc. PODC*, pages 322–331, New York, NY, USA, 2014. ACM. `doi:10.1145/2611462.2611500`.

7 Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proc. PPOPP*, pages 161–170, 2012.

8 Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Proc. Euro-Par*, pages 229–240, Berlin, Heidelberg, 2013. Springer-Verlag. `doi:10.1007/978-3-642-40047-6_25`.

9 Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. *SIGPLAN Not.*, 49(8):343–356, February 2014. Source code available at `https://github.com/logicalordering/trees`. `doi:10.1145/2692916.2555269`.

10 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. PODC*, pages 332–340, New York, NY, USA, 2014. ACM. `doi:10.1145/2611462.2611486`.

11 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proc. PODC*, pages 131–140, New York, NY, USA, 2010. ACM. Source code available at `http://www.cs.toronto.edu/~tabrown/ksts/StaticDictionary5.java`. `doi:10.1145/1835698.1835736`.

12 Amr Elmasry. *Exploring New Frontiers of Theoretical Informatics: IFIP 18th World Computer Congress TC1 3rd International Conference on Theoretical Computer Science (TCS2004) 22–27 August 2004 Toulouse, France*, chapter Adaptive Sorting with AVL Trees, pages 307–316. Springer US, Boston, MA, 2004. `doi:10.1007/1-4020-8141-3_25`.

13 Amr Elmasry and Abdelrahman Hammad. An empirical study for inversions-sensitive sorting algorithms. In *Proc. WEA*, pages 597–601, 2005. `doi:10.1007/11427186_52`.

14 Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. BSDCan*, 2006. URL: `http://www.canonware.com/jemalloc/`.

15 Michael T. Goodrich and Roberto Tamassia. *Algorithm Design and Applications*. Wiley Publishing, 1st edition, 2014.

16 Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proc. PPoPP*, pages 1–10, New York, NY, USA, 2015. ACM. `doi:10.1145/2688500.2688501`.

17 Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. Rank-balanced trees. *ACM Trans. Algorithms*, 11(4):30, 2015. `doi:10.1145/2689412`.

18 Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 2013.

19 Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proc. SPAA*, pages 161–171, New York, NY, USA, 2012. ACM. `doi:10.1145/2312005.2312036`.

20 Mengdu Li. Deletion without rebalancing in non-blocking self-balancing binary search trees. Master's thesis, Dalhousie University, 2016.

21 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proc. PPoPP*, pages 317–328, New York, NY, USA, 2014. ACM. Source code available at `https://github.com/anataraja/lfbst`. `doi:10.1145/2555243.2555256`.

22 Ben Pfaff. Performance analysis of BSTs in system software. In *Proc. SIGMETRICS*, pages 410–411, New York, NY, USA, 2004. ACM. `doi:10.1145/1005686.1005742`.

23 Arunmoezhi Ramachandran and Neeraj Mittal. CASTLE: Fast concurrent internal binary search tree using edge-based locking. In *Proc. PPoPP*, pages 281–282, New York, NY,

USA, 2015. ACM. Source code available at `https://github.com/arunmoezhi/castle`. `doi:10.1145/2688500.2688551`.

24 Arunmoezhi Ramachandran and Neeraj Mittal. A fast lock-free internal binary search tree. In *Proc. ICDCN*, pages 37:1–37:10, New York, NY, USA, 2015. ACM. Source code available at `https://github.com/arunmoezhi/lockFreeIBST`. `doi:10.1145/2684464.2684472`.

25 Riku Saikkonen and Eljas Soisalon-Soininen. Bulk-insertion sort: Towards composite measures of presortedness. In *Proc. SEA*, pages 269–280, 2009. `doi:10.1007/978-3-642-02011-7_25`.

26 Siddhartha Sen and Robert E. Tarjan. Deletion without rebalancing in balanced binary trees. In *Proc. SODA*, pages 1490–1499, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

Proving Opacity of a Pessimistic STM*

Simon Doherty[1], Brijesh Dongol[2], John Derrick[3], Gerhard Schellhorn[4], and Heike Wehrheim[5]

1 **Department of Computing, University of Sheffield, Sheffield, UK**
s.doherty@sheffield.ac.uk
2 **Department of Computer Science, Brunel University, London, UK**
Brijesh.Dongol@brunel.ac.uk
3 **Department of Computing, University of Sheffield, Sheffield, UK**
j.derrick@sheffield.ac.uk
4 **Universität Augsburg, Institut für Informatik, Augsburg, Germany**
schellhorn@informatik.uni-augsburg.de
5 **Universität Paderborn, Institut für Informatik, Paderborn, Germany**
wehrheim@upb.de

Abstract

Transactional Memory (TM) is a high-level programming abstraction for concurrency control that provides programmers with the illusion of atomically executing blocks of code, called transactions. TMs come in two categories, optimistic and pessimistic, where in the latter transactions never abort. While this simplifies the programming model, high-performing pessimistic TMs can be complex. In this paper, we present the first formal verification of a pessimistic software TM algorithm, namely, an algorithm proposed by Matveev and Shavit. The correctness criterion used is *opacity*, formalising the transactional atomicity guarantees. We prove that this pessimistic TM is a refinement of an intermediate opaque I/O-automaton, known as TMS2. To this end, we develop a *rely-guarantee* approach for reducing the complexity of the proof. Proofs are mechanised in the interactive prover Isabelle.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.1.2 Modes of Computation, F.3.1 Specifying and Verifying and Reasoning about Programs, H.2.4 Concurrency

Keywords and phrases Pessimistic STMs, Opacity, Verification, Isabelle, Simulation, TMS2

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.35

1 Introduction

Transactional memory (TM) is a mechanism that provides an illusion of atomicity in concurrent programs. It aims to reduce the burden on programmers of implementing complicated, error-prone synchronization mechanisms. TMs are analogous to database transactions in the sense that both perform a series of updates to data in an all-or-nothing manner — if a transaction succeeds, all its operations succeed, and otherwise, it aborts and all its operations fail. Since the first proposal of a software transactional memory (STM) [28], a number of STM algorithms have been developed [16], and many have made their way into mainstream programming, e.g., the ScalaSTM library, a new language feature in Clojure that uses an STM implementation internally for all data manipulation, the G++ 4.7 compiler (which supports STM features directly in the compiler) and others.

* Doherty and Dongol are supported by EPSRC Grants EP/M017044/1 and EP/N016661/1, respectively. Wehrheim is supported by DFG grant WE2290/8-2.

Intuitively, the purpose of an STM is that the transactions appear to be executed sequentially, i.e., as if their sections of code were protected by locks. However, unlike conventional locking mechanisms, STMs typically allow multiple transactions to be executed concurrently. The desired atomicity property for STMs is *opacity* [14, 3], which requires that all transactions (including aborting transactions) agree on a single sequential history of committed transactions. From a verification perspective opacity proofs represent a challenge beyond correctness conditions such as linearizability [10] due to interleaving at the level of operations as well as transactions.

There are two categories of STM designs: *optimistic* and *pessimistic*. Optimistic STMs assume that conflicts are rare, and when a conflict occurs some transaction is aborted. Transactional aborts cause work to be wasted, and interact badly with operations that are immediately visible outside of a transaction (e.g., consuming input from a stream, or printing to a console). Pessimistic STMs guarantee that no transaction ever needs to abort, thereby avoiding these difficulties. This can be easily achieved at the cost of sacrificing concurrency. For example, it is simple to implement a pessimistic STM that prohibits concurrency between read-only and writing transactions (e.g., by using a read/write lock). However, because conflicts between transactions are rare, overall performance can be improved by allowing read-only transactions to execute concurrently with writers. Supporting this concurrency can involve significant additional complexity, and this additional complexity can make the problem of verifying pessimistic algorithms significantly more difficult.

A number of approaches have so far studied verification of STMs, none of them, however, a pessimistic STM[1]. Here, we present a fully mechanised proof of correctness (i.e., opacity) of a pessimistic STM algorithm, namely that by Matveev and Shavit given in [24]. It poses a significant verification challenge due to the subtle nature of the synchronisation techniques it uses. Particularly difficult is showing that opacity holds when a writing transaction commits (see Listing 4), which may synchronise with another committing writer and all active readers.

Our proof of opacity proceeds via showing refinement (more precisely, a forward simulation) between the STM algorithm and a high-level opaque specification. This follows a general scheme for showing opacity proposed in [9], which used a specification called *TMS2*. Since the development of the TMS2 specification, there has been just one example of its use in a refinement-style verification of opacity [19], where the (simpler) NoRec STM is verified. Here, we present its first application to a pessimistic STM. To this end, both the STM implementation and the abstract specification are given as I/O-automata. This allows us to leverage existing theories within the interactive prover Isabelle [26] for our mechanised proof. The proof of refinement – as usual – requires a large number of invariants, both about the shared and local data of transactions. These invariants need to be shown to be preserved by all operations of all transactions. In order to decrease the complexity associated with such cross-preservation proofs (which are similar to interference-freedom proofs of [27]), we introduce a *rely principle* for transactions into invariance proofs (similar to rely-guarantee reasoning [17]). This provides a systematic way of stating assumptions on transactions as well as proving invariants. The work in this paper shows that this rely principle can make refinement-based proofs scale, even for complex STMs. All of our proofs have been carried out in Isabelle and can be found online [8].

Our presentation of the Matveev-Shavit algorithm is more precise than the original, and resolves certain ambiguities in the original description. In particular, a naive interpretation of the original description would result in an algorithm that was not opaque.

[1] A discussion of related work can be found in the conclusion.

Listing 1 Initialisation.

```
(globalVersion = 1) and (lock = free) and (∀ loc • (version (loc) = 0)) and
(∀ t • (txnVersion(t) = Idle) and (not writerWaiting(t)) and
      (t.wrSet = {}) and (not t.progressSeen))
```

Listing 2 Reader transaction's operations.

```
1:  procedure READ_BEGIN (t)
2:      txnVersion(t) ← Reading                ▷ Inform others that reader t has started
3:      t.temp ← globalVersion                 ▷ Read the global version
4:      txnVersion(t) ← t.temp                 ▷ Set t's txnVersion to stored global version
5:  procedure READ_READ (t, loc)
6:      READ_FROM_MEM (t, loc)                 ▷ Execute as procedure READ_FROM_MEM
7:  procedure READ_COMMIT (t)
8:      txnVersion(t) ← Idle                   ▷ Inform others that reader t has finished
9:  procedure READ_FROM_MEM (t, loc)
10:     if not t.progressSeen then             ▷ Check if committing writer's progress has been seen
11:         if version(loc) = txnVersion(t) then   ▷ Check if loc is potentially being written
12:             await txnVersion(t) ≠ globalVersion   ▷ Wait for committing writer to finish
13:             t.progressSeen ← true          ▷ Inform t's next READ_FROM_MEM that
                                                  the writer's commits have completed
14:     return mem(loc)                        ▷ Read value of loc from the memory
```

The structure of the paper is as follows. In Section 2, we introduce our running example and discuss the choices we made resolving the ambiguities in the original presentation of the algorithm. In Section 3, we introduce I/O automata as a model for opacity and the TMS2 specification. Section 4 develops our methodology based on refinement and rely-guarantee methods for proving opacity for pessimistic STMs. This is applied to the pessimistic STM of Matveev and Shavit [24] in Section 5. Finally, we conclude in Section 6.

2 A Pessimistic STM

In this section, we present the pessimistic STM by Matveev and Shavit [24] (which we will refer to as MSPessTM) where no transaction ever aborts. MSPessTM distinguishes between *read-only* (which perform no writes) and *write* transactions. A read-only transaction starts by calling `READ_BEGIN` , performs a number of `READ_READ` operations, then completes using the operation `READ_COMMIT` (see Listing 2). Similarly, a write transaction starts using operation `WRITE_BEGIN` , performs some number of reads and writes using `WRITE_READ` and `WRITE_WRITE` , respectively (Listing 3), then completes using `WRITE_COMMIT` (Listing 4).

Synchronisation is achieved using shared variables *globalVersion*, *txnVersion*(t) (t being a transaction), etc. as well as transaction-local variables $t.temp$, $t.progressSeen$ etc, which are initialised as in Listing 1. Some variables such as $t.temp$ are unrestricted initially, and hence, do not appear in Listing 1.

(1) MSPessTM uses a *deferred update* strategy: a write transaction t caches all its writes (pairs of locations *loc* and values v) in $t.wrSet$, which are committed to the shared memory when executing `WRITE_COMMIT` .

(2) Readers and writers are synchronised via the counter *globalVersion*. A committing writer will increment *globalVersion* prior to updating *mem* (with writes from its write set) and after these updates are completed. Thus, *globalVersion* is even iff there is a committing writer.

(3) After invoking `WRITE_BEGIN` , a writer transitions through three main phases: *waiting*, *active* and *committing*. There may be multiple waiting writers, but at most one active writer and at most one committing writer. Only the active writer may read or write, and only the committing writer may modify the shared memory. A waiting writer must not have progressed beyond line 18. A writer t becomes active when shared variable $writerWaiting(t)$ becomes *false* (either at line 19, or due to another writer executing line 39), and becomes committing by incrementing *globalVersion* (line 36).

(4) Synchronisation between writers is achieved as follows. Initially, there is neither an active nor a committing writer. Waiting writers compete for the shared *lock* (at line 18), and the winner becomes active. An active writer may enter the "critical section" for a committer by progressing beyond line 32 (which can only happen if there is no committer). The active writer actually becomes committing after executing line 36 (the first increment of *globalVersion*). At this point the writer is both active and committing, and only ceases to be an active writer after executing the code block in lines 37-41. Here, it either makes another writer active (line 39), or if no waiting writers are found, it simply releases *lock* (line 41). Matveev and Shavit refer to the mechanism at line 39 as "passing the baton" because *lock* is effectively transferred from the current active writer to some other waiting writer. Note that because lines 37-41 are executed after the first increment of *globalVersion*, there is no danger of there being more than one committing writer. A committing writer may need to synchronise with reads of another active writer; this is achieved using the mechanism described below.

(5) Synchronisation between readers and writers is the most complex mechanism of the algorithm. To understand this, we first note that from the perspective of a writer, there are two abstract versions of the memory: the *current memory* (which is the value of the shared *mem* variable) and the *new memory* (which is the *mem* updated with all writes in the write set). The synchronisation mechanisms ensure no transaction reads from both current memory and the new memory in an inconsistent manner. Note that a reader can read from the current and new memory without violating consistency if all of the locations read are unchanged (and MSPessTM allows this). Therefore, a writer distinguishes between *current* and *new readers*, which access the current and new memory, respectively.

A writer that has entered the critical section of `WRITE_COMMIT` (i.e., progressed beyond line 33) goes through four distinct phases: *blocking* new readers from accessing changed locations in the new memory (lines 34-35), waiting for *quiescence* from readers of the current memory (lines 42-43), *installing* the current memory (lines 44-45), and *signalling completion* (lines 46-47). Note that lines 36-41 deal with a writer committing then becoming inactive (but still committing) as described above.

A reader t must also wait if t detects that a new memory is being installed as the current memory (lines 10–12), which is true if the version number of the location *loc* that t wants to read is the same as t's transaction version. Such a reader must have read *globalVersion* after the first (but before the second) increment within `WRITE_COMMIT` . On the other hand, a writer waits for all readers that may be accessing the current memory during its quiescence phase. These are determined as non-idle transactions with a version number smaller than the writer's version number.

Listing 3 Writer transaction's begin, read and write operations.

```
15: procedure WRITE_BEGIN(t)
16:     writerWaiting(t) ← true                    ▷ Inform others that writer t has started
17:     while writerWaiting(t) do                  ▷ Check that t has become active
18:         atomic{if lock = free then lock ← taken else goto 17}   ▷ Try to acquire lock
19:         writerWaiting(t) ← false               ▷ t has acquired lock, so can become active
20:     t.temp ← globalVersion                     ▷ Read globalVersion
21:     txnVersion(t) ← t.temp                     ▷ Set txnVersion(t) to the globalVersion read

22: procedure WRITE_READ(t,loc)
23:     if loc ∈ dom(t.wrSet) then
24:         return t.wrSet(loc)                    ▷ If possible return value of loc from own write set
25:     else
26:         READ_FROM_MEM (t, loc)                 ▷ Otherwise return value of loc from the memory

27: procedure WRITE_WRITE(t,loc,v)
28:     t.wrSet ← t.wrSet ⊕ {loc ↦ v}              ▷ Store new value of loc in the write set
                                                   ▷ Notation f ⊕ {x ↦ a} denotes functional override
```

(6) A transaction t executing READ_FROM_MEM must wait for a committing writer at most once (line 12), i.e., after the current writer has committed, a reader will not need to wait for new active writers since any new active writer u is guaranteed to wait for the older reader t when u enters its quiescence phase. Hence, a variable $t.progressSeen$ is used to improve efficiency; once $t.progressSeen$ is set to *true*, future reads may safely read from memory without making further checks.

An intuitive English description of this algorithm is given in [24], but no more precise description is provided. In our work we have developed both a pseudocode description and a formal model of the algorithm. This has allowed us to resolve certain ambiguities in the original presentation. Our presentation is explicit about when the shared variables *globalVersion* and *txnVersion* need to be accessed. A direct implementation of the WRITE_COMMIT operation as presented in [24] would result in a procedure with several superfluous accesses to these variables, causing unnecessary and potentially inefficient memory activity. In the version of the algorithm that we verify, $txnVersion(t)$ is saved in the local variable $t.temp$ at line 30, and then this value is used throughout the operation.

Perhaps more importantly, Matveev and Shavit [24] do not detail how *globalVersion* should be copied into $txnVersion(t)$ at the beginning of a read-only transaction. A naive implementation that implemented this copy non-atomically (by first loading *globalVersion* into a local register and then writing the resulting value into $txnVersion(t)$) would not be opaque. To see this, consider the following execution: (1) some reader t_2 begins and reads *globalVersion* to a local register; (2) an already executing writer t_1 enters its commit operation and passes through the blocking phase (setting the version number of locations in its write set and incrementing *globalVersion*); and finally (3) t_1 checks for quiescence. Assuming that no other reader is currently active, t_1 sees quiescence *as it cannot detect that* t_2 *has already read globalVersion* and exits the loop in lines 42/43. If t_2 next executes the other half of the non-atomic statement, setting $txnVersion(t_2)$ to its old copy of *globalVersion*, we arrive at a situation where the reader t_2 can continue while the writer t_1 copies the values in $t_1.wrSet$ to the shared store. The fact that $txnVersion(t_2)$ is stale means that t_2 will be able to read from locations in $t_1.wrSet$ without becoming blocked at line 12 in READ_FROM_MEM , and may observe inconsistent values. We avoid this problem by using the special *Reading* value to

Listing 4 Writer transaction's commit operation.

```
29: procedure WRITE_COMMIT(t)
30:     t.temp ← txnVersion(t)        ▷ Load t's transaction version into a temporary variable
31:     if even(t.temp) then                                   ▷ Check for a committing writer
32:         await t.temp ≠ globalVersion                       ▷ Wait for committing writer to finish
33:         t.temp ← globalVersion                                    ▷ Re-read global version
34:     for all loc ∈ dom t.wrSet do                  ▷ Prepare to write each loc in t's write set
35:         version(loc) ← t.temp + 1          ▷ Inform other readers that loc is being updated
36:     globalVersion ← t.temp + 1 ▷ Update global version and become a committing writer
37:     if ∃ u • writerWaiting(u) then                          ▷ Check for a waiting writer
38:         choose t.txn ∈ {u | writerWaiting(u)}                ▷ Pick some waiting writer
39:         writerWaiting(t.txn) ← false              ▷ Make the selected waiting writer active
40:     else
41:         lock ← free                                  ▷ Free the lock if no waiting writers seen
42:     for all t.txn ∈ {u | t ≠ u and READING(t, u)} do
43:         await not READING(t, t.txn)) ▷ Wait for each potential reader of current memory
                                          to finish or signal that it will read the new memory
44:     for all (loc, v) ∈ t.wrSet do
45:         mem(loc) ← v ▷ Update memory with new value for each element in the write set
46:     globalVersion ← t.temp + 2                  ▷ Inform others that commits have finished
47:     txnVersion(t) ← Idle                          ▷ Inform others writer t has finished
48: function READING(t, u)
49:     return txnVersion(u) ≠ Idle and txnVersion(u) ≤ t.temp
```

indicate when a beginning transaction is copying *globalVersion* (see line 2 of READ_BEGIN). This technique is used explicitly in Matveev and Shavit's lock-eliding STM [1] to solve essentially the same problem.

3 Modelling STMs as Input/Output Automata

To show that the MSPessTM algorithm satisfies *opacity*, we prove that it is a *refinement* of an intermediate opaque I/O-automaton, known as TMS2. In this section we introduce I/O-automata (IOA) [22] and the TMS2 specification [9], then give examples of the (straightforward) IOA encoding of MSPessTM.

The correctness condition we need to prove about MSPessTM is opacity [14]. Overall opacity guarantees that committed transactions should appear as if they are executed atomically, at some unique point in time, and aborted transactions, as if they did not execute at all. Amongst other things, opacity also guarantees that all reads that a transaction performs are valid with respect to a single memory snapshot.[2] Opacity is formulated as a condition on histories, i.e. sequences of operations of transactions. In the following, we will use the term *trace* to stand for such sequences. When proving opacity of an STM, we thus need to show that all traces an STM allows are opaque.

We do not give a formal definition of opacity here because our proof does not make use of it directly. Instead, our proof strategy leverages two existing results from the literature:

[2] In addition, opacity provides meaning to aborted transactions, but because our case study MSPessTM is a pessimistic algorithm, we elide these details.

the definition of the TMS2 specification by Doherty *et al.* [9], and the mechanised proof that TMS2 is opaque by Luchangco *et al.* [23]. Using these results, trace refinement between MSPessTM and the TMS2 specification proves opacity of MSPessTM.

TMS2 is formalised using input/output automata [22], and hence, our formalisations also use IOA. Moreover, Müller [25] has mechanised the IOA theory (including its simulation rules) in Isabelle, which is now part of the standard Isabelle distribution [26]. As our objective is a mechanised proof using an interactive theorem prover, we thus chose to carry out our proofs within Isabelle. Overall, we obtain a fully mechanised verification of opacity for MSPessTM.

I/O automaton (IOA). An IOA is a labeled transition system P with a set of states Σ_P, a set of actions $acts(P)$ (partitioned into internal and external actions), a set of start states $start(P) \subseteq \Sigma_P$ and a transition relation $trans(P) \subseteq \Sigma_P \times acts(P) \times \Sigma_P$.

The TMS2 specification. The TMS2 specification is given in Figure 1. In IOAs, transitions are typically specified in an operational style: every IOA has a number of variables and transitions are formulated by giving a precondition and an effect of the transition stated in terms of these variables. For each transition, the first line in Figure 1 gives the action name. The transition is *enabled* if all its preconditions, given after the keyword Pre, hold in the current state. The state modifications (effect) of the transition are given as a number of assignments after the keyword Eff. In that, the index t refers to the transaction executing the operation.

The transitions of TMS2 are designed to capture the structural patterns common to most STM implementations defined in terms of read and write operations. The state of TMS2 therefore includes a $status_t$, which is 'notStarted' initially. The status enforces that each transaction must execute TMBegin, then some number of TMRead and TMWrite operations, and finally TMEnd.[3] The status is 'ready' in between reads and writes, and 'committed' after the end of the transaction (i.e., when it has committed). Since operations of different transactions may execute concurrently, the abstract specification splits executing an external operation into several steps, including an *invocation* and a *response*. For example, for TMRead, the external step $inv_t(\texttt{TMRead}(loc))$ represents the invocation when reading from location loc, and $resp_t(\texttt{TMRead}(v))$ represents a read returning with value v. In between, an STM implementing TMS2 must at some time determine the value it reads. In TMS2 this is represented by the internal step $\texttt{DoRead}_t(loc, n)$, which computes v by setting $status_t$ to $readResp(v)$. The internal actions of TMS2 (those prefixed by Do) correspond to the points at which operations "take effect".

Like opacity, TMS2 guarantees that transactions satisfy two critical requirements: (*R1*) all reads and writes of a transaction work with a *single consistent memory snapshot* that is the result of all previously committed transactions, and (*R2*) the *real-time order* of transactions is preserved.

To ensure (*R1*), the state of TMS2 includes $\langle mems(0), \ldots, mems(maxIdx)\rangle$, which is a sequence of all possible memory snapshots. Initially the sequence consists of one element, the initial memory $mems(0)$. Committing writer transactions append a new memory $newmem$ to this sequence (cf. $\texttt{DoCommitWriter}_t$), by applying the writes of the transaction to the last element $mems(maxIdx)$. To ensure that the writes of a transaction are not visible to other transactions before committing, TMS2 (like MSPessTM) uses a *deferred update* semantics:

[3] The full TMS2 specification [9] includes transitions for cancelling and aborting a transaction, which we do not present here, since we do not need them for our pessimistic algorithm.

$inv_t(\texttt{TMBegin})$
Pre: $status_t = \text{notStarted}$
Eff: $status_t := \text{beginPending}$
$beginIdx_t := maxIdx$

$resp_t(\texttt{TMBegin})$
Pre: $status_t = \text{beginPending}$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TMRead}(loc))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doRead}(loc)$

$resp_t(\texttt{TMRead}(v))$
Pre: $status_t = \text{readResp}(v)$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TMWrite}(loc, v))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doWrite}(loc, v)$

$resp_t(\texttt{TMWrite})$
Pre: $status_t = \text{writeResp}$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TMEnd})$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doCommit}$

$resp_t(\texttt{TMEnd})$
Pre: $status_t = \text{commitResp}$
Eff: $status_t := \text{committed}$

$\texttt{DoCommitReadOnly}_t(n)$
Pre: $status_t = \text{doCommit}$
$\text{dom}(wrSet_t) = \varnothing$
$validIdx(t, n)$
Eff: $status_t := \text{commitResp}$

$\texttt{DoCommitWriter}_t$
Pre: $status_t = \text{doCommit}$
$rdSet_t \subseteq mems(maxIdx)$
Eff: $status_t := \text{commitResp}$
$mems := mems \oplus newmem$

$\texttt{DoRead}_t(loc, n)$
Pre: $status_t = \text{doRead}(loc)$
$loc \in \text{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: **if** $loc \in \text{dom}(wrSet_t)$ **then**
$status_t := \text{readResp}(wrSet_t(loc))$
else $v := mems(n)(loc)$
$status_t := \text{readResp}(v)$
$rdSet_t := rdSet_t \oplus \{loc \to v\}$

$\texttt{DoWrite}_t(loc, v)$
Pre: $status_t = \text{doWrite}(loc, v)$
Eff: $status_t := \text{writeResp}$
$wrSet_t := wrSet_t \oplus \{loc \to v\}$

where
$$maxIdx \;\widehat{=}\; max(\text{dom}(mems))$$
$$newmem \;\widehat{=}\; \{maxIdx + 1 \mapsto (latestMem \oplus wrSet_t)\}$$
$$validIdx(t, n) \;\widehat{=}\; beginIdx_t \leq n \leq maxIdx \wedge rdSet_t \subseteq mems(n)$$

Figure 1 The transition relation of TMS2.

writes are stored locally in the transaction t's write set $wrSet_t$ and only published to the shared state when the transaction commits.

All reads in TMS2 must be consistent (i.e., occur from a single memory snapshot), therefore each transaction t keeps track of all its reads from memory in a read set $rdSet_t$. A read of location loc by transaction t checks that either loc was previously written by t itself (**then** branch of $\texttt{DoRead}_t(loc)$), or that all values read so far, including loc, are from the same memory snapshot n, where $beginIdx_t \leq n \leq maxIdx$ (predicate $validIdx(t, n)$ from the precondition, which must hold in the **else** branch). In the former case the value of loc from $wrSet_t$ is returned, and in the latter the value from $mems(n)$ is returned and the read set is updated. The read set of t is also validated when a transaction commits (cf. $\texttt{DoCommitReadOnly}_t$ and $\texttt{DoCommitWriter}_t$). Note that when committing, a read-only transaction may read from a memory snapshot older than $mems(maxIdx)$, but a writing transaction must ensure that all reads in its read set are from most recent memory (i.e., $mems(maxIdx)$), since its writes will update the memory sequence with a new snapshot.

To ensure (*R2*), if a transaction u commits before transaction t starts, then the memory that t reads from must include the writes of u. Thus, when starting a transaction (cf. $inv_t(\texttt{TMBegin})$), t saves the current last index of the memory sequence, $maxIdx$, into a local variable $beginIdx_t$. When t performs a read, the check $validIdx(t, n)$ ensures that that the snapshot $mems(n)$ used has $beginIdx_t \leq n$, which implies that the writes of u are included.

$inv_t(\texttt{TMWrite}(loc, v))$	$write\_write_t(28)$	$resp_t(\texttt{TMWrite})$
Pre: $status_t = Ready$	Pre: $status_t = pending(28, loc, v)$	Pre: $status_t = writeResp$
$t \in Writers$	Eff: $status_t := writeResp$	Eff: $status_t :=$ ready
Eff: $status_t :=$	$wrSet_t :=$	
$pending(28, loc, v)$	$wrSet_t \oplus \{loc \to v\}$	

Figure 2 Transitions of MSPessTM for operation `WRITE_WRITE`.

Encoding MSPessTM as an IOA. The state of the IOA representing MSPessTM contains local variables $status_t$, $wrSet_t$, $progressSeen_t$ and $temp_t$, and shared variable $writerWaiting_t$ and $txnVersion_t$ for each transaction t. Note that $status_t$ (with initial value *NotStarted*) is used to model control flow within each transaction, and hence, does not appear explicitly within the pseudocode in Listings 1-4. The state of the IOA also contains synchronisation variables *globalVersion* (which models the shared global version counter) and *lock* (which models the active writer lock). Finally, the IOA must make the shared memory explicit, thus the state includes two shared variables: *mem* (which maps locations to values) and *version* (which maps each location to a version number).

The IOA models execution by representing each atomic step of the MSPessTM algorithm (typically every line in the algorithm) as single IOA transition. As in TMS2, for each MSPessTM operation, the invocations and responses are external; all other lines of code map to internal actions.

Input arguments to an operation executed by transaction t are modelled as part of the $status_t$ variable. In particular, whenever t is executing an operation, the value of $status_t$ is of the form $pending(pc, \langle input\ values \rangle)$, where pc is the line number of the next step to be executed, and $\langle input\ values \rangle$ are the input arguments. Note that for MSPessTM, $\langle input\ values \rangle$ is *none* for the begin and end operations, a location *loc* for read operations, and a location *loc* and value v for write operations. As an example, Figure 2 shows the three transitions for the `WRITE_WRITE` operation from Listing 3: an invocation action $inv_t(\texttt{TMWrite}(loc, v))$, an internal action $write\_write_t(28)$ (corresponding to line 28 in the algorithm, hence the name), and a response action $resp_t(\texttt{TMWrite})$. The set *Writers* in the precondition of $inv_t(\texttt{TMWrite}(loc, v))$ is used to denote the set of writer transactions; we assume that this set is predetermined in some manner.

4 Verifying opacity as Input/Output automata refinement

We are now equipped with two IOA specifications, one for MSPessTM and one for TMS2. Of the latter we already know that its traces are opaque. Our next objective is to show that MSPessTM refines TMS2 from which opacity of MSPessTM follows. The standard way of verifying a refinement is to use a *forward simulation* between the implementation and the specification, as this allows one to verify the refinement in a stepwise manner. In this section we define *forward simulations*, and then develop a novel method for verifying some of the invariants that one needs as part of the proof of forward simulations. Details of how we apply this to the simulation proof between MSPessTM and TMS2 are given in Section 5.

4.1 Proving opacity via refinement.

To verify that pessimistic STM algorithms are opaque we verify that their IOA representations (in this case MSPessTM) are a *refinement* of TMS2. To define refinement formally we need some definitions. An *execution* of an IOA P is a sequence σ of alternating states and actions, beginning with a state in $start(P)$, such that for all states σ_i except the last,

$(\sigma_i, \sigma_{i+1}, \sigma_{i+2}) \in \mathit{trans}(P)$. A *reachable* state of P is a state appearing in an execution of P. An *invariant* of P is a predicate satisfied by all reachable states of P. A *trace* of P is any sequence of (external) actions obtained by restricting the actions of P to its external actions. The set of traces of P represents P's externally visible behaviour.

Refinement is a property between the visible behaviours of abstract an IOA A and a concrete implementation IOA C. In particular, we say C *refines* A iff every trace of C is also a trace of A. In our setting, each externally visible behaviour consists of a sequence of invoke and response events, including the input/output values of reads and writes.

We let $\mathit{external}(A)$ and $\mathit{internal}(A)$ denote the external and internal actions of IOA A, respectively. Writing $cs \xrightarrow{a}_C cs'$ for $(cs, a, cs') \in \mathit{trans}(C)$, we define:

▶ **Definition 1.** A *forward simulation* from a concrete IOA C to an abstract IOA A is a relation $R \subseteq \Sigma_C \times \Sigma_A$ such that each of the following holds.

Initialisation.

$$\forall\, cs \in \mathit{start}(C) \bullet \exists\, as \in \mathit{start}(A) \bullet R(cs, as)$$

External step correspondence.

$$\forall\, cs \in \mathit{reach}(C), as \in \mathit{reach}(A), a \in \mathit{external}(C), cs' \in \Sigma_C \bullet$$
$$R(cs, as) \wedge cs \xrightarrow{a}_C cs' \quad \Rightarrow \quad \exists\, as' \in \Sigma_A \bullet R(cs', as') \wedge as \xrightarrow{a}_A as'$$

Internal step correspondence.

$$\forall\, cs \in \mathit{reach}(C), as \in \mathit{reach}(A), a \in \mathit{internal}(C), cs' \in \Sigma_C \bullet$$
$$R(cs, as) \wedge cs \xrightarrow{a}_C cs' \Rightarrow$$
$$R(cs', as) \vee \exists\, as' \in \Sigma_A, a' \in \mathit{internal}(A) \bullet R(cs', as') \wedge as \xrightarrow{a'}_A as'$$

The conditions for forward simulation we use here are adapted from Lynch and Vaandrager [21]; our step correspondence conditions use a single abstract step instead of a full sequence as in [21], since this is simpler and sufficient for our proof.

We have proved in Isabelle that the existence of a forward simulation (in the sense given here) is sufficient to ensure trace inclusion (this follows fairly directly from a lemma in the I/O-automaton theory of [25]). Furthermore, a proof that all traces of TMS2 are opaque has been completed in the PVS interactive prover by Luchangco *et al.* [23]. Therefore, proving the existence of a forward simulation from the MSPessTM automaton to TMS2 is sufficent to prove opacity of MSPessTM.

4.2 Proving an Invariant with a Rely

The verification of an actual forward simulation for a specific STM algorithm turns out to depend critically on a complicated invariant. In order to manage this complexity, we have developed, in Isabelle, a scheme that allows us to decompose our invariant into simpler components, and prove that our invariant holds with the help of a *rely condition*. We now describe this scheme in the general case. In Section 5.2, we show how to apply this scheme to the MSPessTM algorithm.

To describe the scheme generically, fix an automaton P whose actions are indexed by transactions from a set T, as in *TMS*2 and *MSPessTM*. That is, we assume $\mathit{acts}(P) \subseteq \mathit{Act} \times T$, for some set Act of action names.

Further, assume we are given a *shared invariant*, $\mathit{sharedI} \subseteq \Sigma_P$, that describes an invariant of P's shared state, and *transaction invariants*, $\mathit{txnI}_t \subseteq \mathit{Act} \times \Sigma_P$, $t \in T$, that describe the relationship between each transaction's local state *upon enabledness of the action* $a \in \mathit{Act}$ and the automaton's shared state. The reason for incorporating actions in transaction invariants is that invariants for transactions typically consists of lots of cases, differentiating between

the different program locations of the transactions. Thus, a transaction invariant $txnI_t(a, s)$ can be read as "the property that holds when transaction t executes a in state s".

Our goal is to prove that the composition of the shared invariant and the transaction invariant is an invariant of P. Formally, we must prove that for all $s \in reach(P)$,

$$sharedI(s) \wedge \forall (a, t) \in acts(P) \bullet txnI_t(a, s) \tag{1}$$

Observe that to prove invariance of property (1), it is sufficient to prove the following four properties:

start. Invariant (1) is true initially, i.e., for all $s \in start(P)$, $sharedI(s)$ and $\forall (a, t) \in acts(P) \bullet txnI_t(a, s)$.

shared. The shared invariant is preserved. Formally, for all states s, s', actions a and transaction t, if $sharedI(s) \wedge txnI_t(a, s)$ and $s \xrightarrow{a,t} s'$ then $sharedI(s')$.

self. Each step of each transaction preserves its own invariant. Formally, for all states s, s', actions a, a' and transaction t, if $sharedI(s) \wedge txnI_t(a, s)$ and $s \xrightarrow{a,t} s'$ then $txnI_t(a', s')$.

cross. Each step of each transaction preserves the invariant of every other transaction. Formally, for all states s, s', actions a, a' and transactions t, u where $t \neq u$, if $sharedI(s) \wedge txnI_t(a, s) \wedge txnI_u(a', s)$ and $s \xrightarrow{a,t} s'$ then $txnI_u(a', s')$.

Unfortunately, the last proof obligation, **cross**, introduces substantial complexity in any verification based on invariants and simulation. To see this, observe that for each step of each transaction t, we must consider the effect of the step on *every* action of the transaction u. If we were to prove the noninterference property directly, we would need to discharge quadratically many proof obligations, one obligation for each pair of actions. We address this issue by introducing a *rely* condition, which describes the possible interference that a transaction may experience during its execution. This method reduces the number of proof obligations from quadratic to linear in the number of actions.

Roughly speaking, a rely condition is a relation over the states of an automaton that must preserve the invariant of each transaction, and that must abstract the transitions of each transaction. We say that a relation $rely_t \subseteq \Sigma_P \times \Sigma_P$, $t \in T$, is a *rely condition* of P, if the following conditions hold.

guar. Each transaction preserves the rely of every other transaction. Formally, for all states s, s', actions a and transactions t, u where $t \neq u$, if $sharedI(s) \wedge txnI_t(a, s)$ and $s \xrightarrow{a,t} s'$ then $rely_u(s, s')$.

rely. The rely must ensure each transaction's invariant. Formally, for all states s, s', actions a and transactions t, if $sharedI(s) \wedge txnI_t(a, s)$ and $rely_t(s, s')$ then $txnI_t(a, s')$.

It is straightforward to see that properties **guar** and **rely** together imply property **cross** above. Thus, we have the following theorem.

▶ **Theorem 2.** *If $sharedI$ and $txnI_t$ for each $t \in T$ satisfy properties **start**, **shared**, and **self**, and there is some $rely_t$ for each $t \in T$ satisfying properties **guar** and **rely**, then for all $s \in reach(P)$, we have $sharedI(s) \wedge \forall (a, t) \in acts(P) \bullet txnI_t(a, s)$.*

This theorem has been formalized and proved in our Isabelle development.

Note that unlike some other rely/guarantee schemes, our rely condition is not required to be reflexive or transitive. In some other schemes, the rely condition describes the interference from any number of environment steps. In our setting, the purpose of the rely condition is to ensure that every step of every other transaction preserves the relying transaction's invariant, so transitivity is unnecessary. As we shall see, for our proof of MSPessTM, the rely condition we use is not transitive.

Of course, standard rely/guarantee approaches also employ a guarantee condition. In a conventional setting, the guarantee condition of a component enables it to be composed with other components whose rely conditions are unknown when the first component is developed. So long as the guarantee of one component implies the rely of the other, the composition is sound. In our setting, no transaction is able to modify any state in the environment of the transactional memory system. Therefore, no transaction is capable of interfering with any other component, except the other transactions. Thus, no explicit guarantee is necessary. We require only that each step of each transaction preserves the rely of every other transaction.

5 Application to MSPessTM

In this section we apply our theory to the verification of the MSPessTM algorithm. As part of the proof, we introduce two auxiliary variables: $CWriter$ and $AWriter$ which keep track of the committing and active writers, respectively. If there is no committing writer, then $CWriter = \bot$, otherwise it has the transaction identifier of the committing transaction (similarly $AWriter$). Initially, we set $AWriter = CWriter = \bot$. $CWriter$ is updated to t when transaction t executes line 36, and to $\bot$ when t executes line 46. $AWriter$ is updated to t either when t acquires the lock at line 18, or when some other (active and committing) transaction sets $writerWaiting_t$ to false at line 39. $AWriter$ is set to $\bot$ when some committing transaction releases the lock at line 41.

5.1 The Simulation Relation

We first define a simulation relation R between the states of MSPessTM and TMS2. We use cs to denote a concrete state (i.e., the state of MSPessTM) and as to denote an abstract state (i.e., the state of TMS2). The value of variable v in cs is given by $cs.v$ (and similarly $as.v$). For reasons of space, it is not possible to describe the entire simulation relation, so we focus our attention on the most challenging and important aspect of our proof: showing that each read operation returns a legal value. It is through read operations that transactions actually observe the state of the memory. The full simulation relation may be viewed online [8].

First, it must be possible to identify particular indices of the memory sequence $mems$ (which is part of as) using the variables of cs. For our refinement proof, we must identify the last element in $mems$ (i.e. $maxIdx$ in Figure 1). Recall that in MSPessTM, each writer increments $globalVersion$ twice when it commits and that $globalVersion = 1$ initially. Thus, the total number of committed write transactions is $\lfloor cs.globalVersion/2 \rfloor$. Also, recall that in the initial state of TMS2 $mems$ has one element, and each committing writing transaction appends a new memory snapshot to $mems$. Our simulation captures this by requiring:

$$\lfloor cs.globalVersion/2 \rfloor = as.maxIdx\,. \tag{2}$$

We must ensure that some step of the MSPessTM read operation corresponds (c.f., Definition 1) to the $\texttt{DoRead}_t(n)$ step of TMS2, for some n. For any transaction t, this abstract read index n is determined by the value of $txnVersion_t$ after t has executed either line 4 or line 21. We let:

$$readIdx_t = \lfloor txnVersion_t/2 \rfloor\,. \tag{3}$$

The index $readIdx_t$ is defined throughout the interval between the response of the transaction t's begin operation, and the point during the commit operation when t sets $txnVersion_t$ to

Idle.[4] Our simulation relation specifies that the read set of each transaction is consistent with $as.mems(readIdx(cs, t))$ throughout the interval over which *readIdx* is defined. This allows us to prove that the precondition of $\texttt{DoRead}_t(n)$ is satisfied over this interval.

We must also show that each concrete read operation returns the correct value (i.e., is consistent with the value returned by the abstract read). That is, we need to show that when a transaction executes line 14 of `READ_FROM_MEM` reading from location *loc*, that the value returned is $as.mems(cs.readIdx_t)(loc)$. To achieve this, our simulation relation must relate the values of the concrete memory to values in the abstract memory sequence. There are three cases to consider. In the first, there is no committing writer and the concrete memory is equal to the latest abstract memory $as.mems(maxIdx)$. In the second, there is a committing writer, t, but the quiescence check has not been passed. Then the abstract $\texttt{DoCommitWriter}_t$ has already added $mem \oplus cs.wrSet_t$ to the end of the abstract memory sequence, so the current concrete memory is now $as.mems(maxIdx - 1)$. If the quiescence check has been passed, then current memory is no longer read by any transaction, so the simulation just needs to state the second property, which holds, even if some elements of the write set have been written to *mem* already. Formally we have:

$$cs.CWriter = \bot \wedge cs.mem = as.mems(maxIdx) \tag{4}$$

$$\begin{array}{l} cs.CWriter = t \wedge (\exists\, u \neq t \bullet \neg\ quiescent(u, cs)) \\ \wedge\ cs.mem = as.mems(maxIdx - 1) \wedge cs.mem \oplus cs.wrSet_t = as.mems(maxIdx) \end{array} \tag{5}$$

$$cs.CWriter = t \wedge (\forall\, u \neq t \bullet quiescent(u, x)) \wedge cs.mem \oplus cs.wrSet_t = as.mems(maxIdx) \tag{6}$$

where $quiescent(u, cs)$ holds, iff *globalVersion* is equal to the *effective transaction version* of transaction u. This version, denoted $effTxnVer(cs, u)$, is equal to $temp_u$, when $txnVersion_u = Reading$, equal to *globalVersion* when $txnVersion_u = Idle$ (an *Idle* transaction is quiescent), and equal to $txnVersion(u)$ otherwise. Note that *quiescent* is equal to the procedure `READING` returning false, except for *Idle* transactions, which need not be checked.

Using (4), (5) and (6) a transaction executing line 14 of `READ_FROM_MEM` (t, loc) returns the correct value in state *cs*, provided that we can guarantee the following two properties.

index. Either $cs.readIdx_t = as.maxIdx$ holds or both $cs.readIdx_t = as.maxIdx - 1$ and $cs.CWriter \neq \bot$ hold.

loc. if $cs.txnVersion_t = cs.globalVersion$ then $cs.CWriter = \bot$ or $loc \notin \mathrm{dom}(ws)$, where $ws = cs.wrSet_{cs.CWriter}$.

The simulation relation, together with **index** implies $cs.mem(loc) = as.mems(cs.readIdx_t)(loc)$ so long as *loc* is not in the write set of any committing transaction, which in turn follows from property **loc**.

Properties **index** and **loc** are proved using the following invariants and transaction invariants of MSPessTM.

inv1. In any state for which $txnVersion_t$ is defined and t is not the committing writer, $globalVersion - 2 \leq txnVersion_t \leq globalVersion$ and $txnVersion_t = globalVersion - 2 \Rightarrow CWriter \neq \bot$.

inv2. $CWriter = \bot$ iff *globalVersion* is odd.

txinv1. Whenever a transaction t is enabled to execute line 14 of the `READ_FROM_MEM` procedure, $txnVersion_t < globalVersion$ or $version(loc) \neq txnVersion_t$.

[4] Recall that $txnVersion_t$ is guaranteed to be in $\mathbb{N}$ throughout this interval.

$$cs'.CWriter = t \Leftrightarrow cs.CWriter = t \tag{7}$$

$$cs.CWriter = t \Rightarrow cs'.globalVersion = cs.globalVersion \wedge \tag{8}$$
$$cs'.mem = cs.mem \wedge cs'.version = cs.version \wedge$$
$$\forall\, u \neq t \bullet (quiescent(cs, u) \Rightarrow quiescent(cs', u))$$

$$cs.AWriter = t \Rightarrow cs'.AWriter = t \wedge cs'.lock = cs.lock \wedge cs'.version = cs.version \wedge \tag{9}$$
$$(cs.CWriter = \bot \Rightarrow cs'.CWriter = \bot)$$

$$cs'.CWriter = cs.CWriter \Rightarrow cs'.globalVersion = cs.globalVersion \tag{10}$$

$$cs'.CWriter \neq cs.CWriter \Rightarrow cs'.globalVersion = cs.globalVersion + 1 \wedge \tag{11}$$
$$(cs'.CWriter = \bot \Rightarrow effTxnVer(cs, t) = cs.globalVersion) \wedge$$
$$(cs'.CWriter \neq \bot \Rightarrow cs.CWriter = \bot)$$

$$cs'.txnVersion_t = cs.txnVersion_t \tag{12}$$

$$\forall\, l \bullet cs'.version(l) = cs.version(l) \vee cs'.globalVersion < cs'.version(l) \tag{13}$$

$$cs'.writerWaiting_t \neq cs.writerWaiting_t \Rightarrow cs.writerWaiting_t \wedge \tag{14}$$
$$\neg\ cs'.writerWaiting_t \wedge cs'.lock = taken \wedge cs.CWriter \neq \bot \wedge$$
$$cs'.AWriter = t \wedge even(cs.globalVersion)$$

Figure 3 Our rely condition is the conjunction of these assertions, along with assertions stating that the local variables of each transaction are not changed.

txinv2. Whenever a transaction t is enabled to execute any of the `WRITE_COMMIT` procedure after line 35 until line 46, we have for all $loc \in \mathrm{dom}(wrSet_t)$, $version(loc) = globalVersion$.

Property **index** follows from invariants **inv1** and **inv2**. Property **loc** follows from invariants **txinv1** and **txinv2**, and we use the generic approach described above to verify these in turn.

5.2 Verifying the invariants for MSPessTM

We now outline how we proved invariants **inv1** and **inv2**. The full invariant is too long to present in this report, but Isabelle source describing the invariant can be obtained from [8]. We focus our attention on the rely condition, and explain how to prove that our key invariants are preserved by this rely. Our rely condition is presented in Figure 3. Note that this rely $rely_t$ states the properties which the transaction t can assume to hold between current and next state while the other transactions $u \neq t$ execute.

We first consider invariant **txinv2**. The antecedent of this invariant is false until t completes the loop at lines 34-35, after which the consequent is true by the effect of that loop. At this point t is the active writer. Properties (9) and (8) of the rely condition describe which aspects of the shared state are stable when a writing transaction is either active or committing. Together, these properties ensure that while $t = AWriter$ or $t = CWriter$, $version$ does not change. Further, properties (9) and (7) ensure that the value of $AWriter$ and $CWriter$ are not changed by another transaction, implying stability of **txinv2**.

We turn now to invariant **inv1**. This invariant is established when the transaction t writes its $temp$ variable into $txnVersion_t$. Properties (12) and (10) of the rely condition ensure that the invariant is preserved over transitions where $cs'.CWriter = cs.CWriter$, because none of the relevant variables are changed. When $cs'.CWriter \neq cs.CWriter$, there are two possibilities, both of which are described by property (11) of the rely. If $cs'.CWriter \neq \bot$, then

$cs.CWriter \neq \perp$, so $cs.txnVersion_t \neq cs.globalVersion - 2$ by invariant **inv1**. The invariant follows easily. If $cs'.CWriter = \perp$, then $cs.txnVersion_t = cs.globalVersion$ by property (11) itself, and hence $cs'.txnVersion_t = cs'.globalVersion - 1$ holds, which preserves the invariant. (Note that property (11) is not transitive because it stipulates that *globalVersion* can only be incremented.)

For reasons of space, we have ignored the question of how we prove the **guar** property of Section 4.2 for our rely. We note only that the transition relation of MSPessTM ensures that when $cs'.CWriter \neq cs.CWriter$ and $cs.CWriter = \perp$, the transition is the step of the transaction $cs'.CWriter$ when it increments *globalVersion* the second time at line 46. MSPessTM has the invariant that at this point, the state is quiescent. The fact that during these steps, $cs.txnVersion_t = cs.globalVersion$ for all t follows from this quiescence.

This proof has been mechanized in Isabelle. This effort took around three weeks of full time work, including building the MSPessTM model and stating and proving the invariant and simulation relation. The proof uses Isabelle theories, including an Isabelle formalisation of the TMS2 automaton, that had already been developed by the authors as part of a larger transactional memory verification project.

6 Related work and conclusions

A number of approaches have so far studied verification of STMs, none of them – however – a pessimistic STM. The proposed techniques range from model checking approaches [12, 13] to interactive proofs [19]. A comprehensive survey of STM verification methods can be found in [18, 6]. Model checking (e.g., [4]) is generally not suitable for our aims of rigorously verifying algorithms against all possible executions. One promising approach is by Guerraoui *et al.* [12, 13], who present a method for model checking opacity using a reduction theorem that lifts opacity for two threads and two variables to opacity for an arbitrary number of threads and variables. However, their specifications do not consider the values that are read or written, and hence, the link to the definition of opacity in [15] is unclear. Moreover, as far as we are aware, the proof of their reduction theorem itself has not been mechanised.

Li *et al.* [20] have verified STM algorithms, however they show correctness against their own abstract specification. Lesani [18] developed a formal proof method for opacity by splitting opacity into a number of other conditions (*markability*). In spirit, this technique is similar to linearization proofs which rely on finding statements in the code which represent linearization points. Very recent work includes [2], which proved the $CaPR^+$ algorithm correct with respect to a notion called *conflict opacity*, which is a subset of opacity. Emmi *et al.* [11] describe a method for inferring invariants in order to prove strict serializability of TM algorithms. This simplifies a crucial task in mechanised proofs; similar techniques could be used for other correctness conditions, including opacity. The verification of TMs in the presence of non-transactional code is studied in [5].

In this paper, we presented a proof of opacity of the pessimistic STM of [24]. Our proof is based on refinement against the TMS2 specification, leveraging existing work that has mechanically verified TMS2 to be opaque [23]. This significantly improves on our previous work that inductively checks opacity [7]. Furthermore, we have developed and used a new generalised reasoning scheme for proving *transaction invariants* via *rely conditions*. The new proof scheme reduces the number of proof obligations from quadratic (with respect to the number of lines of code) to linear complexity.

References

1. Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In M. K. Aguilera, editor, *DISC*, volume 7611 of *LNCS*, pages 297–311. Springer, 2012.
2. A. S. Anand, R. K. Shyamasundar, and S. Peri. Opacity proof for CaPR+ algorithm. In *ICDCN*, pages 16:1–16:4, New York, NY, USA, 2016. ACM. doi:10.1145/2833312.2833445.
3. H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In P. Fatourou and G. Taubenfeld, editors, *PODC'13*, pages 309–318. ACM, 2013.
4. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, pages 37–44, Washington, DC, USA, 2007. IEEE Computer Society.
5. A. Cohen, A. Pnueli, and L. D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 121–134. Springer, 2008.
6. A. Cristal, B. K. Ozkan, E. Cohen, G. Kestor, I. Kuru, O. S. Unsal, S. Tasiran, S. O. Mutluergil, and T. Elmas. Verification tools for transactional programs. In *Transactional Memory*, volume 8913 of *LNCS*, pages 283–306. Springer, 2015.
7. J. Derrick, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim. Verifying opacity of a transactional mutex lock. In *FM*, volume 9109 of *LNCS*, pages 161–177. Springer, 2015.
8. S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim. Isabelle files for a verification of a pessimistic STM algorithm. http://www.informatik.uni-augsburg.de/swt/projects/MSPessTM.html, 2016.
9. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.
10. B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19, 2015.
11. M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. *SIGPLAN Not.*, 45(6):134–145, June 2010.
12. R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memories. In F. van Breugel and M. Chechik, editors, *CONCUR*, pages 21–35. Springer, 2008.
13. R. Guerraoui, T. A. Henzinger, and V. Singh. Model checking transactional memories. *DISC*, 22(3):129–145, 2010.
14. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.
15. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.
16. T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.
17. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
18. M. Lesani. *On the Correctness of Transactional Memory Algorithms*. PhD thesis, UCLA, 2014.
19. M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In M. Koutny and I. Ulidowski, editors, *CONCUR 2012*, pages 516–530, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
20. Y. Li, Y. Zhang, Y.-Y. Chen, and M. Fu. Formal reasoning about lazy-STM programs. *Journal of Computer Science and Technology*, 25(4):841–852, 2010.

21 N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214–233, 1995.

22 N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

23 V. Luchangco M. Lesani and M. Moir. Putting opacity in its place. In *WTTM*, 2012.

24 A. Matveev and N. Shavit. Towards a Fully Pessimistic STM Model. In *TRANSACT*, 2012.

25 O. Müller. I/O Automata and beyond: Temporal logic and abstraction in Isabelle. In J. Grundy and M. Newey, editors, *TPHOLs*, pages 331–348. Springer, 1998.

26 T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

27 S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

28 N. Shavit and D. Touitou. Software transactional memory. *DISC*, 10(2):99–116, 1997.