19th International Conference On Principles Of Distributed Systems

OPODIS'15, December 14–17th, 2015, Rennes, France

Edited by

Emmanuelle Anceaume
Christian Cachin
Maria Potop-Butucaru

LIPICS

LIPIcs – Vol. 46 – OPODIS'15 www.dagstuhl.de/lipics

Editors

Emmanuelle Anceaume
CNRS – IRISA UMR 6074
Rennes
Emmanuelle.Anceaume@irisa.fr

Christian Cachin
IBM Research
Zürich
Christian.Cachin@zurich.ibm.com

Maria Potop-Butucaru
University Paris 6
Paris
Maria.Potop-Butucaru@lip6.fr

ACM Classification 1998
C.2.4 Distributed Systems

ISBN 978-3-939897-98-9

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/978-3-939897-98-9.

Publication date
September, 2016

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.2015.0

ISBN 978-3-939897-98-9 **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

Tutorials

Keynotes

Regular Papers

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The papers in this volume were presented at the 19th International Conference on Principles of Distributed Systems (OPODIS 2015), held from December 14th to December 17th 2015 in Rennes, France. It was organized by CNRS and IRISA and took place in the modern auditorium of Inria on the campus universitaire de Beaulieu.

OPODIS is an open forum for the exchange of knowledge on distributed computing and distributed computer systems. All aspects of distributed systems are within the scope of OPODIS, including theory, specification, design, performance, and system building. With strong roots in the theory of distributed systems, OPODIS covers nowadays the whole range between the theoretical aspects and practical implementations of distributed systems, as well as experimentation and quantitative assessments. This year the topics of interest of OPODIS were:

- Algorithms for distributed systems (static or dynamic) and their complexity;
- Cluster, cloud, grid and high-performance computing;
- Distributed operating systems, middleware, and database systems;
- Communication networks (protocols, architectures, services, applications);
- Cryptographic protocols and security mechanisms;
- Fault tolerance, reliability, availability;
- Internet applications, social systems, peer-to-peer and overlay networks;
- Self-* solutions for distributed systems;
- Mobile and wireless computing and sensor networks;
- Mobile agents and autonomous robots.

We received 91 submissions and each submission was reviewed by at least three members of the Program Committee with the help of external reviewers. This year, the papers of the Program Committee members have been reviewed by at least four members of the Program Committee. Out of the 91 submissions, 31 papers have been accepted as regular papers. The submission and review process was handled using Shai Halevi's web-review software, hosted by the International Association for Cryptographic Research (IACR).

This edition of OPODIS marks the first time where the proceedings appear 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. This form of publication ensures the widest possible dissemination and reduces the cost of accessing the technical contributions for readers.

The Best Paper Award was given to Niloufar Shafiei for the paper "Non-Blocking Doubly-Linked Lists with Good Amortized Complexity".

This year OPODIS had three distinguished invited keynote speakers: Idit Keidar (Technion, Israel), Nicola Santoro (Carleton University, Canada) and Juan Garay (Yahoo Labs, USA). Additionally, the first morning of the conference has been dedicated to the excellent tutorials given by Michel Raynal (Institut Universitaire de France, Université de Rennes 1, France) and Alexander Spiegelman and Idit Keidar (Electrical Engineering Department, Technion, Haifa, Israel).

We would like to thank all authors for submitting their work to OPODIS. We also would like to thank the members of the Program Committee and the external reviewers for their tremendous work and for their availability during the physical Program Committee meeting held at IBM Research – Zürich on November 2–3, 2015.

Organizing this event would not have been possible without the time and the effort of the Organizing Committee, consisting of: Yann Busnel (ENSAI), Romaric Ludinard (ENSAI) as publicity chair and responsible for the website, Elisabeth Lebret (INRIA), and Lydie Mabil (INRIA). 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.

January 2016

Emmanuelle Anceaume, CNRS / IRISA
Christian Cachin, IBM Research – Zürich
Maria Potop-Butucaru, UPMC Sorbonne Universités

Committees

General Chair

Emmanuelle Anceaume, CNRS 6074 – IRISA, France

Program Chairs

Christian Cachin, IBM Research – Zürich
Maria Potop-Butucaru, LIP6

Program Committee

Silvia Bonomi, La Sapienza, Italy
Christian Cachin (program committee co-chair) IBM Research – Zürich, Switzerland
Keren Censor-Hillel, Technion, Israel
Xavier Defago, JAIST, Japan
Shlomi Dolev, Ben-Gurion University of the Negev, Israel
Panagiota Fatourou, FORTH ICS & University of Crete, Greece
Antonio Fernandez Anta, Institute IMDEA Networks, Spain
Christof Fetzer, TU Dresden, Germany
Emmanuel Godard, Université Aix-Marseille, France
Wojciech Golab, University of Waterloo, Canada
Krishna P. Gummadi, MPI SWS Saarbrücken, Germany
Taisuke Izumi, Nagoya Institute of Technology, Japan
Flavio Junqueira, ReScale Limited, United Kingdom
Rüdiger Kapitza, TU Braunschweig, Germany
Aggelos Kiayias, University of Athens, Greece
Fabian Kuhn, University of Freiburg, Germany
Petr Kuznetsov, Telecom ParisTech, France
Dahlia Malkhi, VMware Research, United States
Rui Oliveira, Universidade do Minho, Portugal
Marina Papatriantafilou, Chalmers, Sweden
Fernando Pedone, University of Lugano, Switzerland
Andrzej Pelc, University of Quebec, Canada
Erez Petrank, Technion, Israel
Peter Pietzuch, Imperial College, United Kingdom
Florin Pop, University Politehnica of Bucharest, Romania
Maria Potop-Butucaru (program committee co-chair), Université Paris 6/LIP6, France
Michel Raynal, IUF & IRISA-INRIA Rennes, France
Etienne Rivière, University of Neuchâtel, Switzerland
Luís Rodrigues, INESC-ID, Universidade de Lisboa, Portugal
Matthieu Roy, LAAS-CNRS, France
Alex Schwarzmann, University of Connecticut, United States
Roman Vitenberg, University of Oslo, Norway
Philipp Woelfel, University of Calgary, Canada
Haifeng Yu, National University of Singapore, Singapore

Steering Committee

Marcos K. Aguilera, VMWare, United States
Roberto Baldoni, Sapienza University of Rome, Italy
Giuseppe Prencipe, University of Pisa, Italy
Nicola Santoro, Carleton University, Ottawa, Canada
Marc Shapiro, INRIA, France
Sébastien Tixeuil (steering committee chair), IUF & LIP6-CNRS 7606, France
Maarten van Steen, VU University Amsterdam, Netherlands

Organization Committee

Emmanuelle Anceaume, CNRS / IRISA, France
Yann Busnel (organization chair), ENSAI / CREST / INRIA, France
Elisabeth Lebret, INRIA, France
Romaric Ludinard (publicity chair), ENSAI / CREST, France
Lydie Mabil, INRIA, France
Heverson B. Ribeiro (volunteer), University of Neuchâtel, Switzerland
Nicoló Rivetti Di Val Cerco (volunteer), ENSAI / CREST, France

List of Authors

Mohamad Ahmadi
Department of Computer Science, University of Freiburg, Germany
mahmadi@cs.uni-freiburg.de

Hagit Attiya
Technion, Israel
hagitcs.technion.ac.il

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

Ryan Berryhill
University of Toronto, Canada
ryan@eecg.utoronto.ca

Peva Blanchard
LPD, EPFL, Switzerland
peva.blanchard@epfl.ch

Zohir Bouzid
IRISA, Université de Rennes, France
zohir.bouzid@irisa.fr

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

Claire Capdevielle
Université de Bordeaux, LaBRI, France
claire.capdevielle@labri.fr

Yuval Cassuto
Department of Electrical Engineering, Technion, Israel
ycassuto@ee.technion.ac.il

Armando Castañeda
Instituto de Mathemáticas, UNAM, Mexico
armando.castaneda@im.unam.mx

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

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

Jingshu Chen
Michigan State University, USA
chenji15@cse.msu.edu

Gregory Chockler
CS Department, Royal Holloway, University of London, UK
gregory.chockler@rhul.ac.uk

Artur Czumaj
Department of Computer Science, Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, UK
A.Czumaj@warwick.ac.uk

Ajoy K. Datta
University of Nevada Las Vegas, USA
ajoy.datta@unlv.edu

Peter Davies
Department of Computer Science, Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, UK
P.W.Davies@warwick.ac.uk

Faith Ellen
University of Toronto, Canada
faith@cs.toronto.edu

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

Juan Garay
Yahoo Labs, USA
garay@yahoo-inc.com

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

Seth Gilbert
Department of Computer Science, National University of Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Abdolhamid Ghodselahi
Department of Computer Science, University of Freiburg, Germany
hghods@cs.uni-freiburg.de

Wojciech Golab
University of Waterloo, Canada
wgolab@uwaterloo.ca

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

Magnús M. Halldórsson
Reykjavik University, Iceland
mmh@ru.is

Colette Johnen
Université de Bordeaux, LaBRI, France
johnen@labri.fr

Danny Hendler
Ben-Gurion University, Israel
hendlerd@cs.bgu.ac.il

Maurice Herlihy
Department of Computer Science, Brown University, USA
mph@cs.brown.edu

Stephan Holzer
Massachusetts Institute of Technology (MIT), USA,
holzer@mit.edu

Wei-Lun Hung
University of Texas at Austin, USA
wlhung@utexas.edu

Damien Imbs
Department of Mathematics, University of Bremen, Germany
imbs@math.uni-bremen.de

Hirotsugu Kakugawa
Graduate School of Information Science and Technology, Osaka University, Japan
kakugawa@ist.osaka-u.ac.jp

Nikolaos D. Kallimanis
Foundation for Research and Technology – Hellas (FORTH) & Institute of Computer Science (ICS), Greece
nkallima@ics.forth.gr

Eleni Kanellou
Foundation for Research and Technology – Hellas (FORTH) & Institute of Computer Science (ICS) & Université de Rennes 1, France
kanellou@ics.forth.gr

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

Barbara Keller
ETH Zürich, Switzerland
barbara.keller@tik.ee.ethz.ch

Sven Köhler
Bar-Ilan University, Israel
sven.kohler@biu.ac.il

Fabian Kuhn
Department of Computer Science, University of Freiburg, Germany
kuhn@cs.uni-freiburg.de

Sandeep S. Kulkarni
Michigan State University, USA
sandeep@cse.msu.edu

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

Tobias Langner
ETH Zürich, Switzerland
tobias.langner@tik.ee.ethz.ch

Lawrence L. Larmore
University of Nevada Las Vegas, USA
lawrence.larmore@unlv.edu

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

Toshimitsu Masuzawa
Graduate School of Information Science and Technology, Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Alessia Milani
Université de Bordeaux, LaBRI, France
milani@labri.fr

Adam Morrison
Computer Science Department,
Technion—Israel Institute of Technology
mad@cs.technion.ac.il

Miguel A. Mosteiro
Department of Computer Science, Kean
University, USA
mmosteir@kean.edu

Calvin Newport
Department of Computer Science,
Georgetown University, USA
cnewport@cs.georgetown.edu

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

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

Nathan Pinsker
Massachusetts Institute of Technology
(MIT), USA,
npinsker@mit.edu

Alexandru Popa
Department of Computer Science,
Nazarbayev University, Kazakhstan
alexandru.popa@nu.edu.kz

Anisur Rahaman Molla
Department of Computer Science, University
of Freiburg, Germany
armolla@cs.uni-freiburg.de

Dror Rawitz
Bar-Ilan University, Israel
dror.rawitz@biu.ac.il

Michel Raynal
IRISA, Université de Rennes & Institut
Universitaire de France, France
michel.raynal@irisa.fr

Noam Rinetzky
Blavatnik School of Computer Science, Tel
Aviv University
maon@cs.tau.ac.il

Mohammad Roohitavaf
Michigan State University, USA
roohitav@cse.msu.edu

Nicola Santoro
School of Computer Science, Carleton
University, Canada
santoro@scs.carleton.ca

Vikram Saraph
Department of Computer Science, Brown
University, USA
vsaraph@cs.brown.edu

Christian Scheideler
Paderborn University, Germany
scheideler@uni-paderborn.de

Alexander Setzer
Paderborn University, Germany
asetzer@mail.upb.de

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

David Stolz
ETH Zürich, Switzerland
stolzda@ethz.ch

Thim Strothmann
Paderborn University, Germany
thim@mail.upb.de

Yuichi Sudo
NTT Secure Platform Laboratories &
Graduate School of Information Science and
Technology, Osaka University, Japan
sudo.yuichi@lab.ntt.co.jp

Pierre Sutra
University of Neuchâtel, Switzerland &
Télécom SudParis, CNRS, Université
Paris-Saclay, France
pierre.sutra@telecom-sudparis.eu

Edward Talmage
Parasol Laboratory, Texas A&M University,
College Station, USA
etalmage@tamu.ed

Orr Tamir
Blavatnik School of Computer Science, Tel Aviv University
ortamir@post.tau.ac.il

Mahesh Tripunitara
University of Waterloo, Canada,
tripunit@uwaterloo.ca

Jara Uitto
ETH Zürich, Germany
jara.uitto@tik.ee.ethz.ch

Tonghe Wang
Department of Computer Science,
Georgetown University, USA
tw473@georgetown.edu

Roger Wattenhofer
ETH Zürich, Switzerland
wattenhofer@ethz.ch

Jennifer Welch
Parasol Laboratory, Texas A&M University,
College Station, USA
welch@cse.tamu.edu

Leqi Zhu
University of Toronto, Canada
lezhu@cs.toronto.edu

Signature-Free Communication and Agreement in the Presence of Byzantine Processes

Michel Raynal[*]

Institut Universitaire de France, Paris, France; and
IRISA, Université de Rennes, Rennes, France

Abstract

Communication and agreement are fundamental abstractions in any distributed system. (If the computing entities do not need to communicate or agree in one way or another, the system is not a distributed system!) This tutorial was devoted to the design of such abstractions built on top of signature-free asynchronous distributed systems prone to Byzantine process failures. It is made up of three parts, each devoted to an abstraction and algorithms that implement it.

1998 ACM Subject Classification C.2.4 [Computer-Communication Network] Distributed Systems – distributed applications, network operating systems, D.4.5 [Operating Systems] Reliability – fault-tolerance, F.1.1 [Computation by Abstract Devices] Models of Computation, Computability theory

Keywords and phrases Asynchronous system, Atomic read/write register, Byzantine process, Consensus, Distributed algorithm, Distributed computability, Fault-tolerance, No-duplicity broadcast, Process crash, Reliable broadcast

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.1

Category Tutorial

1 Introduction

1.1 Aim of the tutorial

This tutorial was motivated by the following observations:

- The world is distributed and more and more applications are distributed.
- Asynchronous message-passing systems are more and more pervasive.
- In one way or another, computing entities have to communicate and agree.
- The assumption “no computing entity has a bad behavior” is no longer reasonable.

Its aim was consequently to present basic communication and agreement abstractions which can cope with bad (intentional or not) behavior of a subset of the computing entities.

On the content of this article. This companion text presents the definition of the abstractions addressed in the tutorial. The page limitation does not allow to present the algorithms that implement them. The reader can find them in the corresponding articles or technical reports.

[*] This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 1; pp. 1:1–1:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.2 Underlying computation model

Processes. The computing model is composed of n processes (computing entities), denoted $p_1, \ldots, p_n$, which are sequential and asynchronous (each process progresses at its own speed which is arbitrary and never known by the other processes).

Communication medium. Any pair of processes is connected by a bidirectional communication channel which allows them to send and receive messages. The underlying network is consequently fully connected, allowing a receiver to know which process sent the message it receives. It is assumed to be reliable: there is neither loss, creation, duplication, nor alteration of messages.

Process failure model. A process that executes correctly its local algorithm is said to be *correct*. A process that does not execute correctly its local algorithm is said to be *faulty*. We consider here the severe failure model, where a faulty process can behave arbitrarily, which is called *Byzantine* behavior [21, 30].

A Byzantine process can crash (premature halt), execute arbitrary code, omit to send or receive messages, send erroneous messages, decide not to send a message when it is assumed to do it, send different values to different processes when it is assumed to send them the same value, etc. Byzantine processes can also collude to foil the correct processes.

Let us observe that a correct process can never know if another process is correct or Byzantine. Moreover, a Byzantine process can behave as if it was correct during arbitrary long periods separated by faulty behavior.

On signatures. Digital signatures (based on public key cryptosystems) can be be used to restrain the bad behavior of Byzantine processes. As an example, a Byzantine process which must forward a message signed by a correct process can only forward it or not forward it. It cannot corrupt its content.

Signatures have a computation cost, and require the adversary (here the set of Byzantine processes) to be computationally limited. Moreover, when data are not confidential, signatures are not always necessary to cope with malicious behavior [29, 33]. Hence, the tutorial considered signature-free systems. This allows the adversary to have an unbounded computational power.

Constraint and notation. Let t be the upper bound on the number of Byzantine processes. It is assumed in the following that $t < n/3$. This requirement is necessary to solve the problems in which we are interested here. The corresponding computing system is denoted $\mathcal{BZ\_AS}_{n,t}[t < n/3]$.

2 Reliable Broadcast Abstractions

The aim of a broadcast abstraction is to allow a correct process to send a value to all correct processes, with some provable delivery guarantees. The tutorial considered two of them.

2.1 No-duplicity broadcast

The no-duplicity broadcast (ND-broadcast) was introduced by S. Toueg in [34]. It provides the processes with the operations ND_broadcast() and ND_deliver(), which allows any correct process to broadcast a message (we say "ND-broadcast") and to deliver messages (we say "ND-deliver") while guaranteeing the following properties:

- ND-Validity: If a correct process ND-delivers a message from a correct p_i, then p_i invoked ND_broadcast().
- ND-no-duplicity: No two correct processes ND-deliver distinct messages from the same (correct of Byzantine) process p_i.
- ND-Termination-1: If the sender is correct, all the correct processes eventually ND-deliver its message.

This specification can easily be extended to the case where a process needs to ND-broadcast several messages. In this case a tag must be associated with each message (e.g., a pair $\langle$ sender id, sequence number$\rangle$ and a correct process can ND-deliver only once a message with a given tag (e.g., [7]).

The ND-broadcast abstraction ensures that no two correct processes ND-deliver different messages from the same ND-broadcaster. A message ND-broadcast by any correct process is ND-delivered by each correct process, and no fake message is ND-delivered from a correct process.

Considering an ND-broadcast instance whose sender is a Byzantine process p_k, If a correct process p_i ND-delivers a message m_1 while another correct process p_j ND-delivers a message m_2, we necessarily have $m_1 = m_2$. According to the previous specification, it is nevertheless possible that p_i ND-delivers m_1 while p_j does not deliver a message from p_k. As shown in the tutorial this communication abstraction can be implemented in $\mathcal{BZ}\_\mathcal{AS}_{n,t}[t < n/3]$.

Let us notice that, when process ND-broadcast several (tagged) messages, ND-broadcast does not prevent two correct processes from ND-delivering different sets of messages. These sets may differ in the messages from Byzantine processes: a message can appear in one set and not in another set.

2.2 Reliable broadcast

The reliable broadcast abstraction was introduced in several articles. We consider here a definition close to the given by Bracha [6]. Its associated operations are denoted RB_broadcast() and RB_deliver().

This communication abstraction can be seen as ND-broadcast enriched with an additional termination property, stating that all correct processes RB-deliver the same set of messages, this set including at least all the messages RB-broadcast by the correct processes. More formally, RB-broadcast is defined by the following properties:

- RB-Validity: If a correct process RB-delivers a message from a correct process p_i, then p_i invoked RB_broadcast().
- RB-no-duplicity: No two correct processes RB-deliver distinct messages from the same (correct of Byzantine) process p_i.
- RB-Termination-1: If the sender is correct, all correct processes eventually RB-deliver its message.
- RB-Termination-2: If a correct process RB-delivers a message m from p_i (possibly Byzantine), all the correct processes eventually RB-deliver m from p_i [1].

[1] A similar modular presentation with a separation of Termination-1 and Termination-2, which clarifies the relation between ND- and RB-broadcasts appeared in [8], where the communication abstractions are called consistent broadcast and reliable broadcast, respectively.

This broadcast abstraction can be built in $\mathcal{BZ\_AS}_{n,t}[t < n/3]$, on top of the ND-broadcast abstraction. The algorithm described in [6] generates $O(n^2)$ implementation messages, and, assuming each message takes one time unit, the RB-delivery of a message requires three time units. Moreover, three types of causally-related implementation messages are used in Bracha's algorithm [6]. Recently, a new RB-broadcast algorithm has been proposed [18]. This implementation requires $O(n^2)$ implementation messages (as [6]), but has a smaller time complexity, namely 2 instead of 3.

3 Read/write Register Abstraction

3.1 Read/Write Register

A read/write register is the most basic object encountered in computing science. It provides the processes two operations, denoted read() and write(), which allow the invoking process to obtain the value of the register and assign a new value to the register, respectively.

On the different types of registers. In the presence of concurrency, several processes may concurrently access a register. In such a context, according to the requirement on the values returned by the read invocations, several types of registers can be defined [19, 20], namely safe, regular, and atomic. From a computability point of view, they have the same power in the presence of process crashes (see the textbooks [2, 23, 32]). There is nevertheless a cost to go from safe registers to regular registers, and from regular registers to atomic registers.

Atomic register. We consider here atomic registers. A register is *atomic* [19] (or linearizable [16]) if its read and write operations satisfy the following properties:

- The execution of each operation appears as if it has been executed at a single point of the time line between its start event and its end event,
- No two operations appear at the same point of the time line, and
- Each read returns the last value written before it in the sequence.

3.2 An Implementation in the process crash failure model

A simple and elegant algorithm implementing an atomic register in the asynchronous message-passing model where up to $t < n/2$ may commit crash failure was introduced in [1]. This paper shows also that $t < n/2$ is a necessary requirement for such an implementation. This algorithm is based on the following principles and mechanisms:

- An increasing sequence number is associated with each written value,
- Each process manages a local copy of the register,
- Using request and acknowledgment messages, each write operation updates a majority of local copies,
- Similarly, using request and acknowledgment messages, each read operation obtains $\langle$value, seq. number$\rangle$ pairs from a majority of processes, and returns the value whose sequence number sn id the greatest. Moreover, to ensure atomicity, before returning, the read operation must ensure that a majority of local copies have a value whose sequence number is $\geq sn$.

3.3 Atomic registers in the Byzantine failure model

Parts of this section are from [26].

Single-writer multi-reader register. As it is not possible to constrain the behavior of a Byzantine process, such a process can corrupt any register it can access. Hence, implementing atomic registers in the presence of Byzantine processes is meaningful only if we consider single-writer multi-reader (SWMR) registers, i.e., registers that can be written by a single process, but read by any process. In this way a Byzantine process can only corrupt the registers for which it is the only writer.

Hence, we consider here the construction of an array of SWMR registers $REG[1..n]$, such that $REG[i]$ can be written only by p_i.

Read and write by a Byzantine process. As a Byzantine process can behave arbitrarily, there is no requirement on the value it returns from a read invocation.

As far as the write operation is concerned, the situation is more complicated. A Byzantine process p_k can invoke $REG[k]$.write() as if it was correct. It can also try to modify (by generating appropriate implementation messages) the content of $REG[k]$ without invoking $REG[k]$.write(). If it succeeds, the corresponding modification of $REG[k]$ is considered as if it was produced by an invocation of $REG[k]$.write(). This is because, these two cases cannot be distinguished by the correct processes. Let us also notice that it is not possible to prevent a value written by a Byzantine process from being a value that has nothing to do with the problem to be solved (*fake* value).

Preliminary definitions. The specification of an array $REG[1..n]$ of atomic SWMR registers in a Byzantine failure context is based on the following definitions:

- An *abstract* sequence H_i is associated with each register (intuitively, H_i represents the sequence of values written by p_i in its register $REG[i]$).
- If p_i is correct, let $read[i, j, x]$ denote the execution by p_i of $REG[j]$.read() returning the value of $H_j[x]$.
- Let $write[i, x]$ denote the xth modification of $REG[i]$ (necessarily by p_i):
  - If p_i is correct, $write[i, x]$ is the xth execution of $REG[i]$.write() by p_i.
  - If p_i is Byzantine, $write[i, x]$ is the xth modification of $REG[i]$ by p_i (not necessarily due to an invocation of $REG[i]$.write()).

Specification. Each register of the array $REG[1..n]$ is defined by the following set of properties:

- Termination. If p_i is a correct process, all its invocations of $REG[i]$.write() terminate, and for any j, all its invocations of $REG[j]$.read() terminate.
- Atomicity. Let p_i and p_j be two correct processes, and p_k a correct or Byzantine process.
  - Read followed by write: ($read[i, k, x]$ terminates before $write[k, y]$ starts) $\Rightarrow$ $(x < y)$.
  - Write followed by read: ($write[j, x]$ terminates before $read[i, j, y]$ starts) $\Rightarrow$ $(x \leq y)$.
  - No read inversion: ($read[i, k, x]$ terminates before $read[j, k, y]$ starts) $\Rightarrow$ $(x \leq y)$.

  The first rule states that a process cannot read from the future. The second rule states that a read cannot obtain an overwritten value. The last rule states that, given any register $REG[k]$, sequential read operations concurrent with one or several write operations must respect the sequential order on these writes.

It is easy to show that, from these rules, each register behaves atomically [19][2], i.e., as defined in Section 3.1.

[2] Or is linearizable according to the terminology of [16].

From message-passing to read/write registers. Algorithms implementing atomic registers in $\mathcal{BZ}\_\mathcal{AS}_{n,t}[t < n/3]$ are described in [17, 26]. As shown in [17], $t < n/3$ is a necessary requirement for such an implementation.

Such constructions allow us to execute algorithms designed to run on an SWMR shared memory where at most $t < n/3$ processes may be Byzantine, on top of an asynchronous message-passing system. This is important because designing a Byzantine-tolerant algorithm for a given problem is usually easier in the shared memory context than in the message-passing context. Examples of such algorithms are described in [17].

4 Agreement Abstraction (Consensus)

4.1 The consensus problem

Definition. Consensus is one of the most (maybe the most) important agreement problem of fault-tolerant distributed computing. Its informal statement is extraordinary simple: it requires that all correct processes agree of the same value.

More precisely, assuming that each process proposes a value, the consensus abstraction in a Byzantine failure context, is defined by the following properties:

- C-Termination: Every correct process eventually decides a value.
- C-One-shot: A correct process decides at most once.
- C-Agreement: No two correct processes decide different values.
- C-Validity: If all correct processes propose the same value v, then v is decided.

If only two values can be proposed, consensus is binary. Otherwise it is multivalued. In the following we consider binary consensus. This is not a problem, as there exist algorithms, for asynchronous systems, which solve multivalued consensus on top of binary consensus (e.g., [8, 28, 32] to cite a few).

Impossibility in asynchronous systems. Despite its very simple statement, there is no deterministic algorithm that solves the consensus problem in the presence of asynchrony and failures:

- as soon as $n \geq 2$,
- whatever the communication medium (read/write shared memory or message-passing),
- even if only a single process may fail,
- even if the process failure model is the less severe, namely process crash,
- even if the processes have to agree on a single bit.

This impossibility is a foundation result of fault-tolerant distributed computing. It states a limit of what can be computed in the presence of asynchrony and failures, in the context of read/write communication, or message-passing communication. It has first been established in the context of asynchronous message-passing systems by Fischer, Lynch and Paterson, hence its name "FLP impossibility" result in 1985 [13]. It has then been extended to read/write shared memory systems in 1987 [22].

The intuition that underlies this impossibility lies in the fact that, due to asynchrony, a process is unable to know if another process has crashed or is only very slow (or its incident channels are very slow).

How to circumvent the consensus impossibility. Several approaches have been proposed to circumvent the previous impossibility. We cite here three of them.

- Enrich the system with information on failures. This is the failure detector approach introduced in [11]. This approach allowed to discover the weakest information on failures needed to solve some problems (otherwise impossible to solve). The weakest failure detector to solve consensus in the crash failure model is called Ω [12]. It states that all correct processes must eventually agree on the same leader, which must be one of them.
- Restrict the set of input vectors, where an input vector is an n-size vector, each of its entries containing the value proposed by the corresponding process [27]. The problem consists then in defining the greatest sets of input vectors (each such set is called a condition), such that, given a condition, consensus can be solved if and only if the input vector belongs to the condition.

  Interestingly, a strong connection relating conditions (hence, the consensus problem) with error-correcting codes has been established in [14].
- Enrich the system with randomization. In this case, a process can draw random numbers to face the uncertainty created by the net effect of asynchrony and failures. In this case, the algorithms are no longer deterministic. This approach was introduced to solve consensus in 1983 by M. Rabin [31] and M. Ben-Or [3].

  As far as consensus is concerned, the C-Termination property becomes "Each correct process decides with probability 1". In round-based algorithms (as are consensus algorithms), this property translates as follows

  $$\lim_{r \to +\infty} \text{Proba}[p_i \text{ decides by round } r] = 1.$$

  This is the approach we consider in the following.

4.2 Related works

This section borrows parts (including the table) from [25]. Some of the first randomized consensus algorithms (e.g. [3, 5]) use *local coins* (the values returned to a process by a local coin is not related to the values returned by the local coins of the other processes). As a consequence, they have an expected number of rounds which is exponential (unless $t = O(\sqrt{n})$). As randomized algorithms based on a common coin can have an expected number of rounds which is constant, this paper focuses only on such algorithms.

Common coin. As defined in [9], a *common coin* is a global entity that delivers the same sequence of random bits to each process, each value with probability 1/2.

To obtain a random bit, a process invokes the operation random(), whose rth invocation by any process returns it the bit b_r. Hence, all correct processes obtain the same sequence of random bits b_1, b_2, ..., b_r, etc.

Moreover, it is not possible for Byzantine process to obtain bits in advance, namely, a Byzantine process can obtain the value of b_r only when at least one correct process has started accessing b_r. It follows that a common coin is a strongly synchronized coin (see [9] for the implementation of a common coin).

Randomized asynchronous Byzantine algorithms using a common coin are listed in Table 1 (at the last line, ℓ denotes the length of an RSA signature). All these algorithms, which address binary consensus, are based on asynchronous rounds, and, in each of them, every message carries the number of the round in which it is sent. Hence, when comparing their message size, we do not consider round numbers. We have the following.

Table 1 Cost and constraint of different Byzantine binary consensus algorithms.

Protocol	$n >$	sign.	msgs/round	bits/msg	steps/rd
Rabin [31]	$10t$	yes	$O(n^2)$	$O(1)$	2
Berman Garay [4]	$5t$	no	$O(n^2)$	$O(1)$	2
Friedman Mostéfaoui Raynal [15]	$5t$	no	$O(n^2)$	$O(1)$	1
Bracha [6]	$3t$	no	$O(n^3)$	$O(\log(n))$	9
Srikanth Toueg [33]	$3t$	no	$O(n^3)$	$O(\log(n))$	5
Toueg [34]	$3t$	yes	$O(n^3)$	$O(n)$	3
Canetti Rabin [10]	$3t$	no	$O(n^2)$	poly(n)	9
Cachin Kursawe Shoup [9]	$3t$	yes	$O(n^2)$	$O(\ell)$	2

- The first algorithm is such that $n < 10t$, has an $O(n^2)$ message complexity, and requires signatures.
- The algorithms of the two next lines are such that $t < n/5$, and their message complexity is $O(n^2)$. These algorithms are simple, signature-free, and use one or two communication steps per round, but none of them is optimal with respect to t-resilience.
- The algorithms of the next three lines are optimal with respect to t, but have an $O(n^3)$ message complexity. Moreover, [34] uses signed messages (to prevent message falsification by Byzantine processes), while [6] does not use a common coin, and may consequently execute an exponential number of rounds. Due to their message complexity, these algorithms are costly.
- As far as the last two lines of the table are concerned, we have the following. Both are optimal with respect to the resilience parameter t, and the number of message per round $O(n^2)$, and use signed messages. The algorithm proposed in [10], although polynomial, has a huge bit complexity. The algorithm presented in [9] has two communication steps per round.

4.3 An optimal algorithm

Contrarily to what could be "falsely deduced" from a quick reading at the randomized consensus algorithms cited in Table 1, the formula

$$[\text{quadratic message complexity}] \Rightarrow [(\text{use of signatures})] \vee (t < n/5)]$$

is false.

There is an algorithm (first introduced in [24], and then improved and generalized in [25] to allow the use of a weak common coin) that has the following set of properties:

- The algorithm requires $t < n/3$ and is consequently optimal with respect to t.
- It uses a constant number of communication steps per round.
- The expected number of rounds to decide is constant.
- The message complexity is $O(n^2)$ messages per round.
- Each message carries its type, a round number plus a constant number of bits.
- Byzantine processes may re-order messages sent to correct processes.
- The algorithm uses a weak coin. *Weak* means here that there is a constant probability that, at every round, the coin returns different values to distinct processes.
- Finally, the algorithm does not assume a computationally-limited adversary (and consequently it does not rely on signed messages).

5 Conclusion

The aim of this paper was to provide the reader with the main ideas, and concepts presented in the tutorial given at OPODIS 2015 (which additionally includes corresponding algorithms). The interested reader can obtain a deeper knowledge of the topic by reading papers listed in the bibliography.

Acknowledgments. I want to thank Achour Mostéfaoui with whom I worked a lot on Byzantine failures, and Christian Cachin for a careful reading of this paper and constructive comments, which helped improve its presentation.

References

1 Attiya H., Bar-Noy A. and Dolev D., Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121–132 (1995)

2 Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, Wiley-Interscience, 414 pages (2004)

3 Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing(PODC'83)*, ACM Press, pp. 27–30 (1983)

4 Berman P. and Garay J.A., Randomized distributed agreement revisited. *Proc. 33rd Annual Int'l Symposium on Fault-Tolerant Computing (FTCS'93)*, IEEE Computer Press, pp. 412–419 (1993)

5 Bracha G., An asynchronous $(n-1)/3$-resilient consensus protocol. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 154–162 (1984)

6 Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130–143 (1987)

7 Cachin Ch., State machine replication with Byzantine failures. In *Replication: Theory and Practice*, Springer LNCS 5959, pp. 169–174 (2010)

8 Cachin Ch., Kursawe K., Peztold F., and Shoup V., Secure and efficient asynchronous broadcast protocols. *Proc. 21st Annual International Cryptology Conference (CRYPTO'01)*, Springer LNCS 2139, pp. 524–543 (2001)

9 Cachin Ch., Kursawe K., and Shoup V., Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*,18(3):219–246 (2005, first version: PODC 2000)

10 Canetti R., and Rabin T., Fast asynchronous Byzantine agreement with optimal resilience, *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 42–51 (1993)

11 Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267 (1996)

12 Chandra T., Hadzilacos V., and Toueg S., The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722 (1996)

13 Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382 (1985)

14 Friedman R., Mostéfaoui A., Rajsbaum S., and and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865–875 (2007)

15 Friedman R., Mostéfaoui A., and Raynal M., Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56 (2005)

16 Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492 (1990)

17 Imbs D., Rajsbaum S., Raynal M., and Stainer J., Reliable shared memory abstractions on top of asynchronous Byzantine message-passing systems. *Proc. 21th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'14)*, Springer LNCS 8576, pp. 37–53 (2014)

18 Imbs D. and Raynal M., Simple and efficient reliable broadcast in the presence of Byzantine processes. `http://arxiv.org/abs/1510.06882` (2015), submitted to publication.

19 Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77–85 (1986)

20 Lamport. L., On Interprocess Communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101 (1986)

21 Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401 (1982)

22 Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research,* JAI Press, 4:163–183 (1987)

23 Lynch N.A., *Distributed algorithms.* Morgan Kaufmann Pub., San Francisco (CA), 872 pages (1996)

24 Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, ACM Press, pp. 2–9 (2014)

25 Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of ACM*, 62(4), Article 31, 21 pages (2015)

26 Mostéfaoui A., Petrolia M., Raynal M., and Jard Cl., Atomic read/write memory in Signature-free Byzantine asynchronous message-passing systems. Tech Report 2028, IRISA, University of Rennes, France (2015), `https://hal.inria.fr/hal-01238765`.

27 Mostéfaoui A., Rajsbaum S., and Raynal M., Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922–954 (2003)

28 Mostéfaoui A. and Raynal M., Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Proc. 22nd Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'15)*, Springer LNCS 9439, pp. 194–208 (2015)

29 Mostéfaoui A. and Raynal M., Communication and agreement abstractions in the presence of Byzantine processes. To appear in *IEEE Transactions on Parallel and Distributed Systems* (2016)

30 Pease M., R. Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–234 (1980)

31 Rabin M., Randomized Byzantine generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Press, pp. 116–124 (1983)

32 Raynal M., *Concurrent programming: algorithms, principles, and foundations.* Springer, 530 pages (2013) (ISBN 978-3-642-32026-2)

33 Srikanth T.K. and Toueg S., Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94 (1987)

34 Toueg S., Randomized Byzantine agreement. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pp. 163–178 (1984)

Dynamic Reconfiguration: A Tutorial*

Alexander Spiegelman†[1], Idit Keidar[2], and Dahlia Malkhi[3]

1 **Andrew and Erna Viterbi Dept. of Electrical Engineering, Technion, Haifa, 32000, Israel**
`sashas@tx.technion.ac.il`

2 **Andrew and Erna Viterbi Dept. of Electrical Engineering, Technion, Haifa, 32000, Israel**
`idish@ee.technion.ac.il`

3 **VMware, Palo Alto, USA**
`dahliamalkhi@gmail.com`

Abstract

A key challenge for distributed systems is the problem of *reconfiguration*. Clearly, any production storage system that provides data reliability and availability for long periods *must* be able to reconfigure in order to remove failed or old servers and add healthy or new ones. This is far from trivial since we do not want the reconfiguration management to be centralized or cause a system shutdown.

In this tutorial we look into existing reconfigurable storage algorithms [7, 8, 1, 9, 6, 10]. We propose a common model and failure condition capturing their guarantees. We define a reconfiguration problem around which dynamic object solutions may be designed. To demonstrate its strength, we use it to implement dynamic atomic storage. We present a generic framework for solving the reconfiguration problem, show how to recast existing algorithms in terms of this framework, and compare among them.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases reconfigurable storage algorithms, dynamic object, dynamic atomic storage

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.2

Category Tutorial

1 Introduction

A key challenge for distributed systems is the problem of *reconfiguration*, i.e., changing the active set of servers. Clearly, any production system that provides data reliability and availability for long periods *must* be able to reconfigure in order to remove failed or old servers and add healthy or new ones. The foundations of reconfigurable distributed algorithms are key to understanding and designing dynamic distributed systems.

The study of reconfigurable replication has been active since at least the early 1980s, with the development of group communication and virtual synchrony (see survey in [3]). In recent years, there were several works on reconfigurable (dynamic) storage [7, 8, 1, 9, 6, 10], some of which use consensus for reconfigurations [7, 8] while others assume fully asynchronous

* This work is partially suported by the Israeli Science Foundation.
† Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 2; pp. 2:1–2:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

systems [1, 9, 6, 10]. We feel that the time has come to provide a clear, unifying failure model and a framework for studying the relationship among different solutions.

In this tutorial we define a clear model, and a generic reconfiguration abstraction that can be used as a black-box in dynamic object emulations. In our model, a configuration is defined by sets of changes (such as adding and removing servers). The sequential specification of our reconfiguration problem says that there is a global sequence of configurations, totally ordered in a way that will be defined below. Importantly, it does not require that clients learn every configuration in the sequence, hence it does not necessitate (or imply) consensus. Despite this weak guarantee, we demonstrate the usefulness of our reconfiguration abstraction by implementing a dynamic register on top of it. We also define a failure condition that generalizes the correct majority (of servers) condition from static systems to dynamic ones. On the one hand, our condition is strong enough to be useful, in that we allow servers to fail (or to be switched off) immediately when an operation that removes them from the current configuration returns. And on the other hand, it is sufficiently weak as to allow implementations that preserve the objects' states when the system is reconfigured.

We present a solution for the reconfiguration problem, which is based on the core mechanism in DynaStore [1]. In order to make it generic and simple, we define a *Speculating Snapshot* (SpSn) abstraction, based on [6], which is the core task clients have to solve in order to coordinate. We then show how to recast existing dynamic storage algorithms [7, 8, 1, 9, 6, 10] in terms of this framework. Specifically, we show that the SpSn abstraction can be implemented by extracting the core coordination mechanism from each of these algorithms, (e.g., consensus from RAMBO [7, 8] and weak snapshot from DynaStore [1, 10]). We use this unified presentation to compare their properties and the resulting complexity of reconfiguration.

The remainder of this tutorial is organized as follows: In Section 2 we define the model and failure condition. In Section 3 we define the reconfiguration problem. Then, in Section 4, we introduce the SpSn abstraction, present our generic reconfiguration algorithm, and compare among different SpSn implementations. In Section 5 we demonstrate how to use the reconfiguration algorithm in order to implement a dynamic atomic register. Finally, we conclude in Section 6.

2 Model

A dynamic shared storage system consists of an infinite set Φ of object servers supporting atomic *read-modify-write* (RMW) remote calls by an infinite set Π of clients. Calls may take arbitrarily long to arrive and complete, hence the system is asynchronous. Any number of clients may fail by crashing. The servers may also fail by crashing, but their failures are restricted by the failure model, which we define later. A server or client is *correct* in a run r if it does not fail in r, and otherwise, it is *faulty*.

We study algorithms that emulate reliable shared objects for the clients via dynamic subsets of the servers.

2.1 Configurations

Our definition of configurations is based on [1]; here we extend it to client-server systems. Intuitively, a configuration is a set of included or excluded servers. Fomally, we define *Changes* to be the set $\{+,-\} \times \Phi$. For simplicity we refer to $\langle +, s\rangle$ as $+s$ (and accordingly to $\langle -, s\rangle$ as $-s$). For example, $+s_3$ is a change that denotes the inclusion of server s_3. A *configuration* is a finite subset of Changes, e.g., $\{+s_1, +s_2 - s_2, \text{ and } +s_3\rangle\}$ is a configuration representing

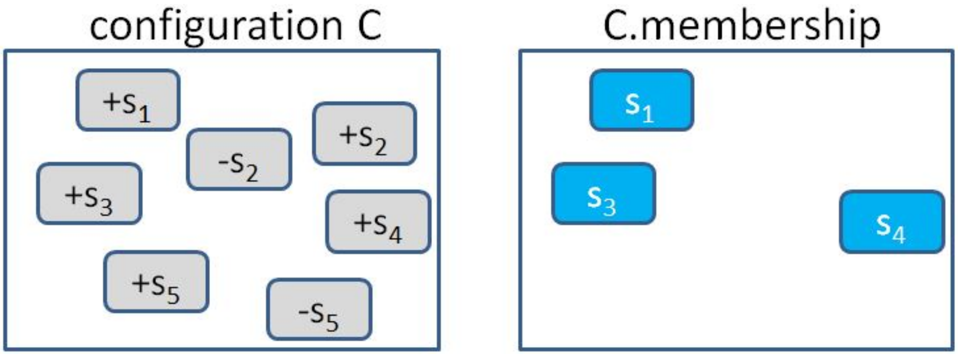


Figure 1 A configuration and its membership.

the inclusion of servers s_1, s_2, and s_3, and the exclusion of s_2. For every configuration C, the membership of C, denoted $C.membership$, is the set of servers that are included but not excluded from it: $\{s| + s \in C \wedge -s \notin C\}$. An excluded server cannot be included again later; in practice, it might join with a different identity. Illustrations of a configuration and its membership appear in Figure 1. Tracking excluded servers in addition to the configuration's membership is important in order to reconcile configurations suggested by different clients. An initial configuration C_0 is known to all clients.

Configuration life-cycle. At different times, configurations can be *speculated*, *activated*, and *expired*. A client can *speculate* a configuration C by issuing an explicit *speculate(C)* event, while configurations are activated and expired implicitly, as we later discuss. A configuration begins its life cycle with its speculation, which occurs after the inclusions and exclusions comprising it have been requested by clients. Then, if it "succeeds", it becomes activated when the system in some sense chooses to make it the new configuration, as we explain in Section 3 below. The newly activated configuration must contain all previously activated ones. (A server is removed by keeping its inclusion and adding its exclusion to the configuration.) A configuration is expired, whether or not it was activated, when a "newer" configuration is activated. Here, we refer to any configuration D as "newer" than C when C does not contain D. The activation of D prevents the future activation of C. Formally:

▶ **Definition 1** (life-cycle). The lifecycle of a configuration C is defined by the following events:

speculate(C): An event that is invoked explicitly by clients.

activate(C): An event that is triggered automatically by certain client operations, as defined below.

expire(C): An event that occurs automatically and immediately when some configuration $C' \nsubseteq C$ is activated.

We later use these notions in order to define the failure model as well as the reconfiguration problem.

Shared register emulation. For every configuration C, as long as a majority of $C.membership$ is alive, clients can use ABD [2] to simulate a collection of atomic read/writer registers on top of the servers in $C.membership$. Thus, we define:

▶ **Definition 2** (availability). A configuration C is *available* if majority of the servers in $C.membership$ are correct.

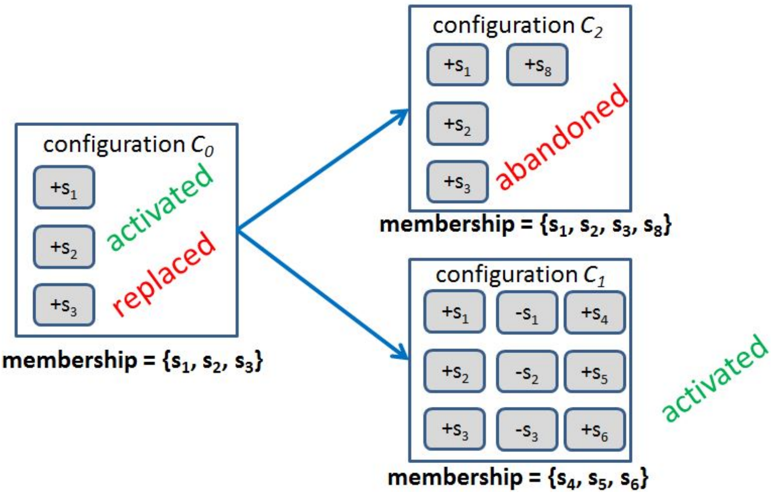


Figure 2 Example of activation and expiration. First client c_a speculates configuration C_1 in C_0, while client c_b speculates configuration C_2 in C_0, client c_a misses C_2 and activates C_1. Note that $C_0, C_2 \not\supseteq C_1$, and thus, according to our definition, both C_0, C_2 are expired and are allowed to be unavailable. Therefore, the excluded servers in C_1 (s_1, s_2, and s_3) are no longer needed for liveness and can be safely switched off.

For simplicity, we henceforth use this abstraction, and have clients invoke atomic read/write operations on shared registers in a given configuration. Note that if a configuration is unavailable, pending reads and writes to and from the configuration's registers might never return. We next define a failure condition that specifies which configurations must be available.

Failure condition. There are infinity many servers in the system, and they cannot all be "alive" from the beginning. A *speculate* event indicates when we expect a configuration to become available. And now, the question is when a configuration can become unavailable. In our failure model we require *reconfigurablity*, which means that once we succeed to activate a new configuration C, every server s that is excluded in C (i.e., $-s \in C$) can fail or be switched off. To this end, we define the following failure condition, which is sufficiently weak so as to allow reconfigurablity.

▶ **Definition 3** (failure condition). If configuration C is speculated and not expired, then C is available.

Recall that when a configuration C is activated then every configuration $D \not\supseteq C$ is expired. Intuitively, we can think of the activated configuration as *chosen*, of activated ones that are expired as *replaced*, and of speculated and not activated ones that are expired as *abandoned*. Figure 2 shows how our failure condition satisfies reconfigurablity.

2.2 Discovering available configurations

Since configurations can be expired and become unavailable, we cannot guarantee termination of operations on emulated registers in expired configurations. Moreover, clients trying to access the shared object would need to access newer configurations in order to complete their operations. For example, a client that arrives after C_0 is expired and does not know of any newer configuration may hang forever because C_0 can already be unavailable. It is easy to see that this limitation is inherent in every model that separates clients from servers and requires reconfigurability (or any other failure model that allows failures of servers in an old configuration).

Therefore, clients have to somehow be notified about new activated configurations. Note that a speculated configuration C can become unavailable only when some configuration $C' \nsubseteq C$ is activated. Thus, when a client tries to access an unavailable configuration C, we want to help the client find an activated configuration.

To this end, we envision that the system would use some *directory* service, or *oracle*, that stores information about configurations and ensures only a relaxed consistency notion[1]. A client which activates a configuration C informs the directory that C is activated. A client which tries to access a configuration D simultaneously posts queries to the directory about D. If the directory service sees that a configuration C has been activated such that D does not contain C, it reports back to the client that D has been expired by an activated configuration C. It is important to note that two clients that post queries to the directory about D can get different responses about activated configurations, and we do not require any eventual consistency properties on these reports. Hence, this service is weak and does not provide consensus.

Such a directory service can be easily implemented in a distributed manner with multiple directory servers. A client informs its local directory server about new activated configurations, and the server broadcasts it to all the other directory servers. When a client posts a request about an expired configuration D to its local directory server, then eventually the server will learn about some configuration that could have expired D and return it to the client.

In order to keep the model simple, we avoid using an explicit directory. Instead, we take an abstraction from [6]:

▶ **Definition 4** (oracle)**.** When a client accesses an unavailable configuration C it gets an error message referencing some activated configuration $C' \nsubseteq C$.

3 Reconfiguration Problem

We want to allow clients that access a shared object to change the subset of servers over which it is emulated. To this end, we define a reconfiguration abstraction, which has one operation, *reconfig*. A *reconfig* operation gets as parameters a configuration C and a proposal $P \subset Changes$. Intuitively, *reconfig*(C, P) is a request to reconfigure the system from configuration C to a new configuration reflecting the changes in P. It returns two values. The first is a configuration C' which is either $C \cup P$ or a superset of it, i.e., $C' \supseteq C \cup P$, where C' may contain additional, concurrently proposed changes. It also returns a set S consisting of all the configurations that were speculated during the operation, and in particular, $C' \in S$. We assume that C is a configuration that was previously returned by some *reconfig* operation (note that this is an assumption on usage). By convention, we say that *reconfig*(C_0, C_0) returns $\langle C_0, \{C_0\}\rangle$ at time 0.

Next we need to determine when configurations are activated. Since one of the purposes of reconfiguration is to allow administrators to switch off removed servers, we want to make sure that *reconfig* leads to the activation of a new configuration, which in turn expires old ones. However, since at the moment when a configuration is activated all preceding

[1] In today's practical settings, it is reasonable to presume that some global directory is available, e.g., DNS.

configurations may become unavailable, we want to allow clients to transfer object state residing in the old configuration to the new one. Clearly every distributed service maintains *some state* on behalf of clients. Thus, when reconfiguring, we need to be careful to not lose this state. We want to define a generic way, without any specific knowledge of the higher level service, to make clients aware of a reconfiguration that is about to happen so they will be able to transfer state to it. Therefore, in our model, a configuration C is not necessarily immediately activated when *reconfig* returns it. Instead, when *reconfig* returns C, our model allows clients to transfer state from previous configurations to C. Only when a client calls *reconfig*(C, P) (for some P), and returns C (indicating no further changes) does C become activated. Formally:

▶ **Definition 5** (activation). A configuration C is *activated* when *reconfig*(C, P) returns $\langle C, S\rangle$ for some S and P for the first time.

Note that by the convention, the initial configuration C_0 is activated at time 0.

Sequential specification. The reconfiguration abstraction is linearizable with respect to the sequential specification consisting of the three properties we now define. Briefly, the idea is that we require that *reconfig* return to all clients configurations that are totally ordered by containment. Importantly, we do not require *reconfig* to return to every client the entire totally-ordered sequence. Rather, we allow clients to "skip" configurations.

First, we require that every change in a speculated configuration was previously proposed:

- **Validity:** A *reconfig* operation *rec* returns $\langle C, S\rangle$ s.t. for every $C' \in S$ for every $e \in C'$, $\exists$*reconfig*(C'', P') that is either *rec* or precedes it s.t. $e \in P'$.

Second, we require that new configurations contain all previous ones:

- **Monotonicity:** If $\langle C, S\rangle$ is returned before $\langle C', S'\rangle$, then $C \subseteq C'$.

Note that it is possible for one client to activate C' after C, while another client reconfigures C to another configuration C''. The third property uses speculation to ensure that the second client is aware of configuration activated by the first:

- **Speculation:** If a configuration $C' \supset C$ is activated before *reconfig*(C, P) returns $\langle C'', S\rangle$, then $C' \in S$.

Liveness. In addition, if the number of invoked *reconfig* operations is bounded, we require:

- **Termination:** Every reconfig operation invoked by a correct client eventually returns.

It is easy to see that in this model (servers and clients are separated), if there is an unbounded number of reconfigurations' invocations, then a correct client may forever chase after an available configuration and never be able to communicate with the servers, and thus, never complete its operation[2].

4 Reconfiguration Solution

In this section we give a generic algorithm for the reconfiguration problem. In Section 4.1 we define the Speculating Snapshot (SpSn) abstraction, which is the core task behind

[2] In [11], we show that even in a model where clients are not distinct from servers, and only clients that are part of the last activated configuration's membership are allowed to invoke operations, we cannot guarantee progress in case of infinite number of reconfiguration even if we can solve consensus in every configuration.

Table 1 Possible SpSn outputs.

Client	Input	Output
c_1	$\{+s_1\}$	$\{\ \{\{+s_1\}\},\ \{\{+s_1\},\{+s_2\},\{+s_3\}\}\ \}$
c_2	$\{+s_2\}$	$\{\ \{\{+s_1\}\},\ \{\{+s_1\},\{+s_3\}\}\ \}$
c_3	$\{+s_3\}$	$\{\ \{\{+s_1\}\},\ \{\{+s_1\},\{+s_2\}\},\ \{\{+s_1\},\{+s_2\},\{+s_3\}\}\ \}$
c_4	$\{\}$	$\{\}$

the algorithm. In Section 4.2 we use the SpSn abstraction in order to present a generic reconfiguration algorithm. In Section 4.3 we show different ways to implement SpSn by recasting existing algorithms of atomic dynamic storage in terms of SpSn.

4.1 SpSn abstraction

SpSn (based on [6]) is the core task clients solve in configurations in order to coordinate configuration changes. It is a multi-input, multi-output task: each client inputs its proposal P by calling *SpSn(P)*, and the output is a set of sets of proposals proposed by different clients. We will use SpSn for *reconfig* by proposing changes, and each of the sets returned by SpSn will be speculated. SpSn is emulated in a given configuration C, and its invocation in C with proposal P is denoted $C.SpSn(P)$. Like other emulated objects, C.SpSn can return an error message with some newer activated configuration if C is unavailable. Within an available configuration C, the SpSn task is defined as follows:

- **Non-triviality:** If $P \not\subseteq C$, then *SpSn(P)* returns a non-empty set.
- **Intersection:** There exists a non-empty set of proposals that appears in all non-empty outputs.

An example of possible SpSn outputs appears in Table 1.

4.2 Generic algorithm for reconfiguration

In this section we show a simple generic algorithm for the reconfiguration problem, which is based on the dynamic storage algorithm presented in DynaStore [1]. We use one SpSn task in every configuration. When clients call $C.SpSn$ with different proposals, they may receive different configurations in return, which in turn leads them to speculate different configurations.

The pseudocode of the algorithm appears in Algorithm 1. The idea is to track the configurations that clients speculate, and try to merge them into one configuration that will reflect them all. In addition, we want to make sure that later *reconfig* operations will be aware of this configuration in order to guarantee monotonicity. We call this process *traverse* [1] because it is a traversal of a configuration DAG (see Figure 2 above) whose nodes are the speculated configurations and there is an edge from a configuration C to a configuration C' if some client receives C' in the output of $C.SpSn$.

Initially, the set *ToTrack* contains only the input configuration C in which the operation started, *proposal* is the union of C and the input proposal P, and the set *speculation* is empty. During the reconfig operation, a client repeatedly takes the smallest configuration in *ToTrack*, speculates it, adds it to the *speculation* set, and proposes *proposal* in its SpSn. Then, if the configuration is available, it adds the output from SpSn (set of configurations) to *ToTrack* (in order to track them later), and adds the union of all the changes in configurations returned from SpSn to *proposal*. Note that each client traverses a different sub-graph of the DAG of

Algorithm 1 Generic algorithm for reconfiguration

```
operation $reconfig(C, P)$
    $ToTrack \leftarrow \{C\}$
    $proposal \leftarrow P \cup C$
    $speculation \leftarrow \{\}$                                ▷ set of speculated configurations
    while $ToTrack \neq \{\}$ do
        $C' \leftarrow \operatorname{argmin}_{C'' \in ToTrack} (|C''|)$        ▷ smallest configuration (in number of changes)
        $speculate(C')$
        $speculation \leftarrow speculation \cup \{C'\}$
        $ret \leftarrow C'.SpSn(proposal)$
        if $ret = \langle$"error"$, C_a\rangle$ then                ▷ $C'$ is expired - restart from $C_a$
            $speculation \leftarrow \{\}$
            $ToTrack \leftarrow \{C_a\}$
        else
            $ToTrack \leftarrow (ToTrack \cup \{\bigcup_{e \in E} e \mid E \in ret\}) \setminus \{C'\}$
        $proposal \leftarrow \bigcup_{e \in ToTrack} e \cup proposal$
    return $\langle proposal, speculation\rangle$
end
```

configurations during its *reconfig* operation. However, since SpSn guarantees intersection, the DAGs of different clients that start in the same configuration intersect (see example in Figure 3). A *reconfig* operation completes when *ToTrack* is empty. The last configuration in *ToTrack* is the configuration where the DAGs merge.

While traversing a DAG, if some configuration is unavailable, a client receives an error message with a newer activated configuration C_a, and starts over from C_a. Note that since we assume a bounded number of reconfigurations, clients with pending operations will (1) eventually reach an available (forever) configuration, and (2) propose the same configuration. Therefore, all the operations eventually complete.

4.3 Recasting existing algorithms in terms of SpSn

In this section we look into existing algorithms and extract their core mechanism for implementing SpSn. As noted above, we assume a shared memory abstraction in every configuration, and as long as the configuration is available, clients can access its read/write registers. Therefore, we implement SpSn in shared memory, and when a configuration C becomes unavailable, pending SpSn invocations in C return an error message with some active configuration.

We make use of a *collect* operation, which returns a set of values of an array of registers. This operation can be implemented by reading the registers one by one, or by opening the ABD abstraction and collecting an entire array of registers in a constant number of communication rounds. In our complexity analysis, we count a collect as one operation.

The SpSn implementations differ in their complexity (number of operations) and in the number of configurations clients traverse in the generic reconfiguration algorithm (i.e., their DAG size). Denote by m the total number of *reconfig* operations, where n of them propose unique changes. As we will see in Section 5, when emulating a dynamic atomic register, read and write operations invoke *reconfig* without proposing changes (they call *reconfig* in order to ensure they execute in the up-to-date configuration), while reconfigurations of the register

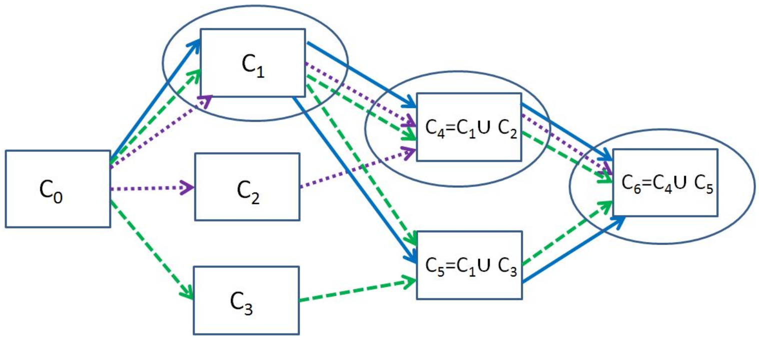


Figure 3 Clients c_a, c_b, and c_c start *reconfig* in configuration C_0. Client c_a traverses the solid blue arrows, c_b traverses green dashed arrows, and c_c traverses purple dotted arrows. For example, c_a first invokes $C_0.SpSn(C_1)$ and receives $\{C_1\}$. Next it invokes $C_1.SpSn(C_1)$ and receives $\{C_4, C_5\}$. Then, it invokes $C_4.SpSn(C_6)$ and $C_5.SpSn(C_6)$ and receives $\{C_6\}$ from both them. Finally, it invokes $C_6.SpSn(C_6)$, receives $\{C_6\}$, and returns (no more configurations to track) C_6 together with a set consisting of the configurations in its DAG. Each of the clients traverses a different sub-graph but since SpSn guarantees intersection, their traversals intersect and eventually merge. The circled configurations are those where the sub-graphs intersect.

Table 2 Comparison among SpSn implementations extracted from existing dynamic storage algorithms

Algorithm	SpSn Cost	DAG size	reconfigurable	rely on consensus
RAMBO [7, 8]	O(1)	n	yes	yes
DynaStore [1]	O(1)	$min(mn, 2^n)$	yes	no
SmartMerge [9]	O(1)	n	no	no
Parsimonious SpSn [6]	O(n)	n	yes	no

propose changes. Table 2 compares the different SpSn implementations described in detail below.

4.3.1 RAMBO

RAMBO [8, 7] was the first to implement a dynamic atomic register with asynchronous read/write operations. The main idea is to use consensus to agree on the reconfigurations, while *read/write* operations asynchronously read from all available configurations and write to the "last" one. We now show how to use consensus in order to implement SpSn. The pseudocode appears in Algorithm 2. It uses a shared array arr where client c_i writes to $arr[i]$. We assume that arr is dynamic: only cells that are written to are allocated.

Consider a client c_i that proposes P in SpSn of configuration C. If $P \nsubseteq C$ (meaning that the client has new changes to propose), it proposes P in C's consensus object, and writes the decision value to its place in arr. Otherwise, it does not invoke consensus and writes nothing to arr. In both cases, it returns the set of sets of values collected from arr. (Only written cells are collected). Note that this set is either empty, in case no changes were proposed, or contains exactly one set of one configuration (the one agreed in the consensus). Therefore, the SpSn non-triviality and intersection properties are preserved.

Note also that clients invoke consensus only if they propose new configurations. Thus, if we use this SpSn in the register emulation of Section 5.2, we preserve the RAMBO property of asynchronous read/write operations.

Algorithm 2 Consensus-based SpSn; protocol of client c_i in configuration C

```
operation $SpSn(P)$
    if $P \nsubseteq C$ then
        $arr[i] \leftarrow C.consensus(P)$
    $ret \leftarrow collect(arr)$
    return $\{\{C'\} \mid C' \in ret\}$
end
```

Algorithm 3 Weak snapshot-based SpSn; protocol of client c_i in configuration C

```
operation $SpSn(P)$
    if $P \nsubseteq C$ then
        $arr[i] \leftarrow P$
    $ret \leftarrow collect(arr)$
    if $ret = \{\}$ then
        return $ret$
    else
        $ret \leftarrow collect(arr)$
        return $\{\{C'\} \mid C' \in ret\}$
end
```

4.3.2 DynaStore

DynaStore [1] was the first algorithm to solve dynamic storage reconfiguration in completely asynchronous systems (without consensus). It observes that clients do not have to agree on the next configuration: different clients can return different configurations, as long as we make sure that if one client writes in some configuration, others will traverse this configuration, read its value, and transfer it to the new configuration they return.

The core mechanism behind the coordination of the algorithm is the weak snapshot abstraction. We now show how to use it in order to implement SpSn. The pseudocode of client c_i implementing *SpSn(P)* in configuration C appears in Algorithm 3. Again, we use an array arr. If c_i proposes a new configuration ($P \nsubseteq C$), then it writes P into its register in arr. Otherwise, it writes nothing. Then it collects the registers in arr. If the collect is empty, it returns $\{\}$, otherwise it collects again and returns the obtained set.

Note that both properties of SpSn are preserved. First, non-triviality is satisfied since if $P \nsubseteq C$ then the client writes to its register and so the collects cannot return an empty set. Second, intersection is satisfied since all the clients that return non-empty sets get a non-empty set in the first collect, and thus collect again. Therefore, in the second collect they all get the first value that is written (this value appears in all outputs).

Note that while the DAG size obtained in the generic algorithm by using consensus for SpSn is exactly n, with a weak snapshot-based SpSn, the DAG can be much bigger. Without consensus, clients can write (propose) different configurations in SpSn's array (arr), and learn different subsets of proposals (from the *collect*), which in turn leads to different proposals being written in the next tracked configuration's SpSn.

With n is unique proposals, there are 2^n possible configurations that can be speculated and traversed. But since clients propose in each SpSn during the traverse the union of all the configurations and proposals they previously traversed, every client proposes at most n different configurations during its traverse. Therefore, the worst-case DAG size is $min(nm, 2^n)$.

4.3.3 SmartMerge

SmartMerge [9] is very similar to DynaStore, but it uses a pre-computation in order to reduce the DAG size to n. Before starting the generic algorithm, clients participate in an external lattice agreement service [4], in which they input their proposals and each receives a set of proposals s.t. all the outputs are related by containment. Then, they take the output of the lattice agreement and use it as their proposal in the generic *reconfig* algorithm (Algorithm 1).

Notice that by ordering the proposals by containment, SmartMerge reduces the total number of configurations that can be speculated and traversed (i.e., the DAG size) to n. However, this solution assumes that the lattice agreement service is available forever, and since it is not a dynamic service, the servers emulating it cannot fail or be switched off. Therefore, SmartMerge is not reconfigurable.

4.3.4 Parsimonious SpSn

Parsimonious SpSn [6] uses multiple rounds of a mechanism similar to commit-adopt [5]. Similarly to SmartMerge, it relies on containment in order to reduce the DAG size, but does not use an external service for it, and thus the solution is reconfigurable. Instead, all the configurations in the sets returned from the commit-adopt-based SpSn are related by containment.

In order to achieve the containment property, parsimonious SpSn pays in the SpSn's complexity. Instead of O(1) as in other implementations, the SpSn complexity here is O(n). More details can be found in [6].

5 Dynamic Atomic Register

The reconfiguration problem can be used as an abstraction in order to implement many dynamic atomic objects on top of it. Here we demonstrate it by presenting a protocol for dynamic atomic register. In Section 5.1 we define the dynamic atomic register object, and in Section 5.2 we present an algorithm that implements it in our model.

5.1 Definition

We consider a dynamic atomic multi-writer, multi-reader (MWMR) register object emulated by a subset of the servers in ϕ, from which any client can *read* or *write* values from a domain $\mathbb{V}$. The sequential specification of the register requires that a *read* operation return the value written by the latest preceding *write* operation, or $\perp$ if there is no such write. In addition, the object exposes an interface for invoking *reconfiguration* operations that allow clients to change the set of servers emulating the register.

A *reconfiguration* gets as a parameter a set of changes $\mathit{Proposal} \subset \mathit{Changes}$ and returns a configuration C s.t. (1) C is activated, (2) $C \supseteq \mathit{Proposal}$, and (3) C is subset of changes proposed by clients before the operation returns.

We assume that there is a bounded number of *reconfiguration* operations, and require that every operation by a correct client eventually returns.

5.2 Solution

We present an algorithm for a dynamic atomic register, which uses the reconfiguration problem abstraction (*reconfig*). The pseudocode appears in Algorithm 4.

Algorithm 4 Dynamic Atomic register emulation

```
Local variable:
    $version,\ tmp \in \mathbb{N} \times \mathbb{V}$ with selectors $ts$ and $v$, initially $\langle 0, v_0 \rangle$
    $C_{cur} \subset Changes$, initially $C_0$

operation $reconfiguration(Proposal)$
    $check\text{-}config(Proposal, \bot, REC)$
    return $C_{cur}$
end

operation $read()$
    $check\text{-}config(\bot, \bot, READ)$
    return $version.v$
end

operation $write(v)$
    $check\text{-}config(\bot, v, WRITE)$
    return ok
end

procedure $check\text{-}config(P, v, op)$
    $\langle C, S \rangle \leftarrow reconfig(C_{cur}, P)$
    repeat
        for each configuration $C' \in S$ do          ▷ read in all speculated configurations
            $tmp \leftarrow C'.readVersion()$
            if $tmp \neq error(*) \wedge\ tmp.ts > version.ts$ then          ▷ newer version found
                $version \leftarrow tmp$
        if $op =\ WRITE$ then
            $version \leftarrow \langle version.ts + 1, v \rangle$
        $C.writeVersion(version)$
        $C_{tmp} \leftarrow C$
        $\langle C, S \rangle \leftarrow reconfig(C_{tmp}, \{\})$          ▷ activate $C$ or find newer configuration
    until $C = C_{tmp}$
    $C_{cur} \leftarrow C$          ▷ $C$ is activated
```

The main procedure used by all operations, (*read*, *write*, and *reconfiguration*), is *check-config*(P, v, op), where P is the reconfiguration proposal ($\bot$ in case of *read* or *write*), $v \in \mathbb{V}$ is the value to write ($\bot$ in case of *read* or *reconfig*), and *op* is the operation type (READ, WRITE, or REC). The procedure manipulates two local variables: C_{cur}, which stores the last activated configuration returned from a *reconfig* operation, and *version*, a tuple consisting of a value v and its timestamp ts. All operations first call *check-config*(P, v, op), and then return according to the operation type: A *write* returns ok, a *read* returns $version.v$, and a *reconfiguration* returns C_{cur}.

In order to emulate the dynamic register we use an idea that was first introduced in RAMBO [7], and later adopted by DynaStore [1]. The idea is to read a version from each configuration that other clients may have written to, and then write the most up-to-date version (associated with the highest timestamp) to the configuration we want to activate and return. To this end, we start *check-config* by calling *reconfig*$(C_{cur}, \mathit{Proposal})$, which returns $\langle C, S\rangle$. Next, we read the version from every configuration returned in S, and write the latest/newer version into C. In case of a *write* operation we write a version consisting of v and a new timestamp (higher than all those we read). Otherwise, we write back the version with the highest timestamp we read.

Note that before we can return, we need to validate that future operations will not miss our version. Therefore, after we write the version, we check if there are new activated configurations. To this end, we call *reconfig*$(C, \{\})$ (line 27). If the operation returns $\langle C', S'\rangle$ where $C' = C$, it is guaranteed that no one "moved forward" before we wrote our version to C, and every later operation will not miss our version. Note that in this case, by our definition, C is activated and older configurations can become unavailable. This does not pose a problem, since the state of the object (the up-to-date version) has already been transferred to the new configuration. Otherwise, if the operation returns $\langle C', S'\rangle$ where $C' \neq C$, we repeat the above process for C' and S'. Notice that since we assume a bounded number of *reconfiguration* operations, it is guaranteed that every *check-config*, and thus every operation performed by a correct client eventually returns.

In order to read and write versions from configurations we assume that every configuration emulates a version object that has two functions: $\mathit{readVersion}()$ and $\mathit{writeVersion}(\mathit{version})$. A $\mathit{readVersion}$ invoked in configuration C simply returns C's version. A $\mathit{writeVersion}(\mathit{version})$ overwrites C's version if it has a higher timestamp, and returns ok. Again, if C is unavailable, the operations return error messages. Note that $\mathit{readVersion}()$ can be implemented by the first phase of ABD [2], and $\mathit{writeVersion}$ by the second.

6 Conclusion

Reconfiguration is a key challenge in implementing distributed dynamic shared objects, and in particular, in distributed dynamic storage. Clearly, any long-lived shared object emulated on top of fault-prone servers must be able to reconfigure in order to remove failed or old servers and add healthy or new ones.

In this tutorial we first defined a clear model for studying reconfiguration. We defined a failure condition that provides reconfigurability, that is, allows a server to fail or be switched off immediately when it is no longer part of the current active configuration's membership. Then, we encapsulated a reconfiguration problem that is on the one hand implementable in asynchronous systems satisfying our failure condition, and on the other hand can be used as an abstraction for implementing many dynamic shared objects. Next, we presented a (simple) general framework for solving the reconfiguration problem, and showed how existing dynamic storage algorithms [7, 8, 1, 9, 6, 10] can be recast in terms of this framework. In order to do so, we defined, (based on [6]), a core task called SpSn, which clients solve in order to coordinate. We demonstrated how to extract different algorithms' coordination mechanisms in order to implement this task. This allowed us to compare the different algorithms.

Finally, we demonstrated the power of the reconfiguration abstraction by presenting a simple algorithm for dynamic atomic storage on top of it.

Acknowledgements. We thank Eli Gafni for his contribution to the SpSn abstraction definition, and Christian Cachin and Yoram Moses for insightful comments.

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 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

3 Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):1–43, December 2001.

4 Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.

5 Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152. ACM, 1998.

6 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.

7 Seth Gilbert, Nancy Lynch, and Alex Shvartsman. Rambo ii: Rapidly reconfigurable atomic memory for dynamic networks. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 259–259. IEEE Computer Society, 2003.

8 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.

9 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.

10 Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, LADIS'10, pages 22–26, New York, NY, USA, 2010. ACM. doi:10.1145/1859184.1859191.

11 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. *CoRR*, abs/1507.07086, 2015. URL: http://arxiv.org/abs/1507.07086.

Time to Change: On Distributed Computing in Dynamic Networks[*]

Nicola Santoro

School of Computer Science, Carleton University, Ottawa, Canada
santoro@scs.carleton.ca

Abstract

In highly dynamic networks, topological changes are not anomalies but rather integral part of their nature. Such networks are becoming quite ubiquitous. They include systems where the entities are mobile and communicate without infrastructure (e.g. vehicles, satellites, robots, or pedestrian smartphones): the topology changes as the entities move. They also include systems, such as peer-to-peer networks, where the changes are caused by entities entering and leaving the system, They even include systems where there is no physical mobility at all, such as social networks. A vast literature on these dynamic networks has been produced in many different fields, including distributed computing. The several efforts to survey the status of the research and attempts to clarify and classify models and assumptions, have so far brought more valuable bibliographic data than order and clarity. Goal of this note is to ask questions that might bring author and readers to start to clarify some important research aspects and put some order in a sometimes confusing field. The focus here is entirely on distributed computing, specifically on its deterministic aspects.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation, F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases distributed computing, dynamic networks, time-varying graphs, mobile agents

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.3

Category Keynote

1 Introduction

Computing in networked environments has been one of the core research areas and concerns of distributed computing. The structure of such environments is modeled as a simple graph, where the nodes correspond to the computational entities and the edges to existing communication links between pairs of entities.

While assuming the network structure to be static, the research soon focused on the study of topological changes in order to model faults and failures occuring in real distributed systems. Examined in the context of *fault-tolerance*, the changes were however considered a small scale phenomenon, limited and localized. The possibility of extensive changes recurring in the lifetime of the system has been object of study within the field of *self-stabilization*: incorrect computations are allowed to take place in a period of instability; it is however assumed that the instability stops, at least long enough, so that the computation can eventually produce correct results. None of these studies can deal with systems where the topology is subject to

[*] This work was supported by NSERC (Canada) under the Discovery Grants program.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 3; pp. 3:1–3:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

extensive changes that can occur everywhere and possibly never stop, systems where changes are not anomalies but rather integral part of their nature.

Such systems do indeed exist and are becoming quite ubiquitous. Such are for example systems where the entities are truly mobile and can communicate without infrastructure (e.g. vehicles, satellites, robots, or pedestrian smartphones): an edge exists between two entities if they are within communication range; the topology changes (possibly dramatically) as the entities move. In these systems end-to-end connectivity does not necessarily hold, the network might be always disconnected, still communication may be available over time and space making broadcast, routing, computations feasible. These networks have been extensively studied from engineers uder the names of delay-tolerant or challanged networks. In addition to these *ad-hoc wireless mobile networks*, the same level of dynamic changes occur in systems where there is no explicit communication, such as swarms of *autonomous mobile robots*: each robot sees the positions of the robots within its visibility range, and based on these positions, it computes a destination and moves there; the topology of the visibility graph changes during the execution of protocol. It can also occur in systems where the changes are caused by entities entering and leaving the system, such as *peer-to-peer networks*. It can even occur in systems where there is no physical mobility at all, such as *social networks*.

Due to the abundance of different contexts where these highly dynamic networks arise and their importance, a vast literature in many different fields has been produced, including by distributed computing researchers, focusing on one or another aspect of these systems. Significant efforts have been made to model and formally describe the aspects under examination; not surprising, the "lexicon" is rather confusing, with the same object being given different names (e.g., “temporal distance” introduced in [23], has been subsequently called “reachability time” in [55], “information latency” in [64], and “temporal proximity” in [65]) and sometimes the same name being used to define different classes of objects (e.g., “temporal graphs" defined in [65] *vs* the later use in [74]).

There have been several efforts to survey the status of the research, attempting to clarify and classify models and assumptions and research results (e.g. the recent [19, 57, 67, 74]), in particular the monumental effort by Holme [56]. From the distributed computing viewpoint, these efforts have so far brought more valuable bibliographic data and interesting information than order and clarity. Goal of this note is to ask questions that might bring author and readers to start to clarify some important research aspects and put some order in a sometimes confusing field.

2 What and How to Represent?

There are many popular ways to represent dynamic networks: *temporal networks*, *evolving graphs*, *multiplex networks*; as discussed below, they are actually all equivalent and equally limited by the restrictive assumptions they make.

A less constrained general mathematical formalism that describes many different types of dynamic networks is the one offered by TVG (for *time-varying graph*) introduced in [28] and described next.

2.1 TVG

TVG is a simple model that includes the other commonly used representations as instances, plus it allows to express other (possibly more complex) computational dynamic systems.

In this model, the dynamic system is described as a *time-varying graph* $\mathcal{G} = (V, E, \mathcal{T}, \psi, \rho, \zeta)$, where V is the set vertices or nodes, representing the system entities (e.g., vehicles, robots) and $E \subseteq V \times V(\times L)$ is the set of (directed or undirected) edges representing connections (e.g. communication, contact, relation, link,...) between pairs of entities; edge can be labeled, the labels in L are domain-specific (e.g., intensity of relation, type of carrier, ...) and possibly multi-valued (e.g. <*satellite link; bandwidth of 4 MHz; encryption available;...*>).

The system exists for a contiguous interval of time $\mathcal{T} \subseteq \mathbb{R}$ called *lifetime.* The system is said to be *limited* if $\mathcal{T}$ is closed, *unlimited* otherwise; in both cases, it is generally assumed that the system has a beginning, which occurs at time $t = 0$.

The dynamics of the system is specified by the *node presence* function, $\psi : V \times \mathcal{T} \to \{0, 1\}$, and the *edge presence* function, $\rho : E \times \mathcal{T} \to \{0, 1\}$, where $\psi(x, t) = 1$ (resp., $\rho(e, t) = 1$) $\Longleftrightarrow$ node $x \in V$ (resp., edge $e \in E$) is present at time $t \in \mathcal{T}$.

Finally, the function $\zeta : E \times \mathcal{T} \to \mathbb{R} \cup \{\perp\}$, is the *latency* (or duration, delay, ...) of the connection. So, for example, $\zeta((x, y), t) = d$ may indicate that a message from x to y, if sent at time t, will arrive at time $t + d$; or that the traversal by a mobile agent of edge (x, y), if started at time t, is completed at time $t + d$. On the other hand, $\zeta((x, y), t) = \perp$ indicates that, if starting at t, there is not enough time to use the connection (eg, send a message, perform a traversal, ...).

An important concept is that of a *snapshot* of the system at time $t \in \mathcal{T}$, denoted by $G(t) = (N(t), E(t))$, where $N(t) = \{x \in N : \psi(x, t) = 1\}$ and $E(t) = \{e \in E : \rho(x, t) = 1\}$ are the nodes and edges present at time t. The *footprint* of the system is just the aggregate graph of all footprints: $G = (V, E)$. It is assumed that the footprint is connected; otherwise, the system is considered composed of separate non-interacting dynamic systems, each with a connected footprint. Observe that connectivity of the footprint G has no implication on the connectivity of any of the snapshots $G(t)$.

The TVG model can be naturally *extended* by adding any number of (possibly temporal) functions on the nodes (e.g., $f_i : V \times \mathcal{T} \to F_i$) and/or on the edges (e.g., $h_j : E \times \mathcal{T} \to H_j$) to appropriate domains, to describe specific system features (e.g., cost, weight, energy, ...).

2.2 Synchronous Systems

The model can be also *restricted* by imposing assumptions. The most common restriction is by assuming a *discrete synchronous* system: the changes in the system occur at discrete time steps (i.e., $\mathcal{T} \subseteq \mathbb{N}$), and its dynamics is fully described as a sequence of synchronous rounds.

For discrete synchronous systems, a TVG (or aspects of it) can be equivalenty and conveniently expressed in other ways.

For example, a compact representation is by listing for each edge e the set $I(e)$ of all the contiguous intervals of time when e was present. Indeed the couple $\mathcal{I} = (N, I)$, where $I = \bigcup_{e \in E} I(e)$, is a common definition of the class of discrete synchronous systems, and it is known in the literature as *temporal networks* [57]. Note that temporal neworks do not consider node presence and more importantly they have no latency; in other words they describe discrete synchronous systems with *instantaneous contacts.*

Another popular representation for discrete synchronous systems is by considering the sequence $S(\mathcal{G}) = < G(0), G(1), G(2), \ldots, G(t), \ldots >$ of all[1] the snapshots of $\mathcal{G}$. Indeed any

[1] A more succint representation is to consider only the snapshots where a topological change occurs.

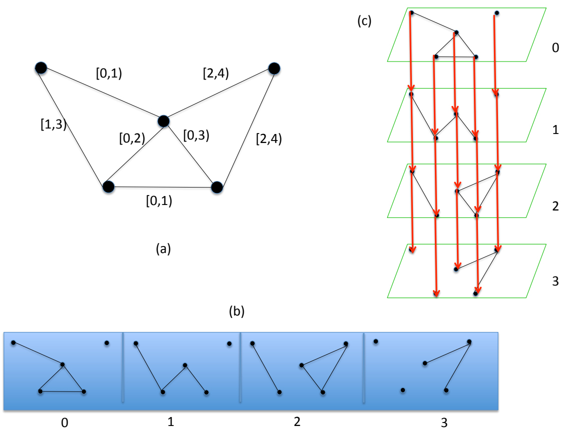


Figure 1 A discrete synchronous limited TVG represented as (a) a temporal network, (b) an evolving graph, (c) a multiplex network.

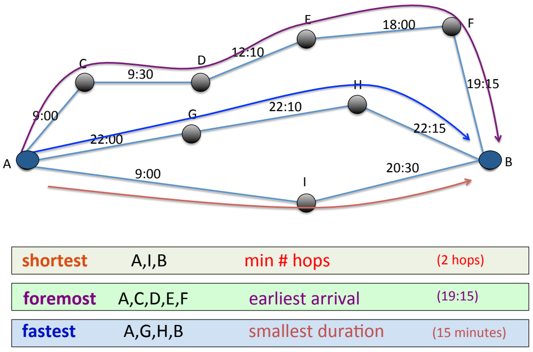
sequence of static graphs $S = < G_0, G_1, G_2, \ldots, G_t, \ldots >$, called *evolving graph* [44], can be seen as the sequence of snapshots of a unique TVG $\mathcal{G}$ (once the latency function has been defined). The idea of representing a dynamic graph as a sequence of static graphs was first suggested in [52]; the proposal of using a sequence of graphs to model discrete synchronous systems was made by [89] in the context of social networks, and by [44] in the context of ad-hoc mobile networks. The evolving graph representation is perhaps the most commonly used for discrete synchronous systems (often without references and under new names). As mentioned, the latency has to be specified and added to this representation to make it equivalent to that of a (discrete synchronous) TVG.

Given the sequence of snapshots $S(\mathcal{G}) = < G(0), G(1), G(2), \ldots, G(t), \ldots >$, consider the multy-layer graph $\mathcal{M}(S)$ obtained connecting $G(t)$ to $G(t+1)$ by adding an edge from each node in $G(t)$ to the same node in $G(t+1)$ (if present in both snapshots). Clearly this multi-layer graph captures the same information as the evolving graph $S(\mathcal{G})$; thus, if enanced with the specification of the latency, it becomes computationally equivalent to the TVG $\mathcal{G}$. Such a multy-layer graph $\mathcal{M}$, called *multiplex network*, is a commonly used represention of a discrete synchronous systems (for recent survey see [67]).

2.3 Journeys and Distances

A crucial concept in dynamic networks is that of *journey*, the dynamic equivalent of "walk" in static graphs. More precisely a journey is a sequence $\mathcal{J} =< (e_1, t_1), (e_2, t_2), \ldots, (e_k, t_k) >$, where $< e_1, e_2, ..., e_k >$ is a walk in G, $\rho(e_i, t_i) = 1$ and $t_i + \zeta(e_i, t_i) \leq t_{i+1}$. That is, the walk edges are present in the graph at the appropriate times with the latency long enough so each edge can be traversed in time. Time t_1 is the start time of the journey $\mathcal{J}$, and $t_k + \zeta(e_k, t_k)$ is the time it ends, denoted by $start(\mathcal{J})$ and $end(\mathcal{J})$ respectively. Journeys could actually be infinite, in which case they have no end.

Depending on whether or not there are time gaps in the walk, we can distinguish between two types of jurneys: a journey $\mathcal{J} =< (e_1, t_1), (e_2, t_2), \ldots, (e_k, t_k) >$ is *direct* if, for all $1 \leq i < k$, $t_i + \zeta(e_i, t_i) = t_{i+1}$, i.e., there is no waiting before traversing any edge; otherwise is *indirect*. This distinction is sometimes relevant because there are dynamic networks where buffering is not supported (and thus only direct journeys are allowed). Some interesting



shortest	A,I,B	min # hops	(2 hops)
foremost	A,C,D,E,F	earliest arrival	(19:15)
fastest	A,G,H,B	smallest duration	(15 minutes)

Figure 2 Three types of minimal journeys; in this example latency is 0.

differences between allowing and not allowing waiting have been recently established [25, 58]. In the following, we assume that waiting is allowed, and thus make no distinction between direct and undirect journeys.

A finite journey is a walk over time from a source to a destination and therefore has not only a *topological* length $|\mathcal{J}|$, defined as the number of edges in the walk, but also a *temporal* length $||\mathcal{J}|| = end(\mathcal{J}) - start(\mathcal{J})$ defined as the time elapsed to perform the walk.

This gives rise to distinct definitions of *distance* in a time-varying graph $\mathcal{G}$:

- `shortest distance`: $d(u,v,t) = Min\{|\mathcal{J}| : \mathcal{J} \in \mathcal{J}_{(u,v)} \wedge start(\mathcal{J}) \geq t\}$ (i.e., min hop);
- `fastest distance`: $\delta(u,v,t) = Min\{||\mathcal{J}|| : \mathcal{J} \in \mathcal{J}_{(u,v)} \wedge start(\mathcal{J}) \geq t\}$ (i.e., min duration);
- `foremost distance`: $\partial(u,v,t) = Min\{end(\mathcal{J}) : \mathcal{J} \in \mathcal{J}_{(u,v)} \wedge start(\mathcal{J}) \geq t\}$ (i.e., ends first);

where $\mathcal{J}_{(u,v)}$ denotes the set of all journeys from u to v.

Indeed, for all all the classical measures of static graphs and networks (diameter, degree, eccentricity, centrality, etc.) there exist (one or more) temporal counterparts (temporal diameter, temporal degree, temporal eccentricity, temporal centrality, etc.).

3 What to Investigate?

3.1 Dead or Live, Centralized or Distributed?

Most of the existing algorithmic investigations and results assume global a-priori knowledge of the system; that is, the entire graph $\mathcal{G}$ is given as an input to the computation. In other words, they consider dynamic networks that are (not only discrete synchronous but also) *limited*; the investigation is *post-mortem*, i.e. after the system has ended its limited life-time (and no more data is being produced); and the computation is off-line, totally *centralized*. That is, they are *centralized investigations of dead systems*. This is for example the case of [16, 23, 44, 48, 53, 59, 64, 78, 73], and in particular of all the investigations on data previously collected from real dynamic networks (e.g., [62, 63, 66, 82, 84, 85, 92]).

The interest of this note is however on *distributed computations in dynamic networks*. This means that the computation is distributively performed *inside* the time-varying graph. In other words, the system is *live*; its lifetime $\mathcal{T}$ is *unlimited* (as far as the computation is concerned); and the computation is *decentralized* and *localized*.

The lifetime $\mathcal{T}$ of $\mathcal{G}$ is divided in three parts: the instantaneous *present*, $\hat{t}$; the limited *past*, $past(\hat{t}) = \{t \in \mathcal{T} : 0 < t < \hat{t}\}$; and the unlimited *future*, $future(\hat{t}) = \{t \in \mathcal{T} : t > \hat{t}\}$.

Each computational entity (node, web site, vehicle, ...) operates in the present, and is aware only of the *local events*, i.e., topological changes in which it is involved; it might remember its past (if it has enough memory); however, it does not know the future.

3.2 Who is in Control?

To understand how to deal with the future, it is important to understand the relationship between the changes occuring in the system and the computation performed by the entities. To ask what is causing the system changes is important but it does not necessarily clarify the nature of the relationship. For example, in ad-hoc wireless mobile networks, it is the movement of the entities that causes the topological changes; similarly, in robotic swarms or in mobile sensor networks, the changes are generated by the movement of the entities. Apparently, in all these systems, the cause of changes is the same: the entities' movements; there is however a fundamental difference between the former and the latter systems.

In the latter, the movement of an entity is influenced by the computation: where an entity goes next (and thus what new edges are being formed) is determined by the protocol; in other words, the *computation generates the graph.* This means that we (algorithm designers) could program the entities so to construct graphs with specific properties. This indeed is what happens when we design protocols that allow autonomous mobile robots with limited visibility to arrange themselves in space so to form a specific geometric pattern, or mobile sensors to spread over a territory so to homogenously cover it. We shall call these types of situations as of *controlled generation* of the graph.

Totally different is the first type of systems: the movements are independent of the computation (e.g., broadcast, routing, etc) performed by the entities. More precisely, the computation has no control over the topological changes of the system. We call this type of situation as of an *uncontrolled generation* of the graph; in this situation, we actually envision the changes as caused by an adversary operating against the computation.

It is on this type of systems that we focus in the rest of this note.

3.3 What Problems?

In the live systems we consider, the computational entities operate in a decentralized and localized matter in an ever changing scenario, without control over the changes. The research investigations on these systems have been both intensive and extensive, carried out mainly within the engineering community.

The distributed computing focus has been on a variety of classical problems, such as *information propagation*: routing, multicast, broadcast, gossip, etc. (e.g. [4, 5, 13, 14, 26, 27, 31, 32, 33, 35, 42, 54, 83, 93]); *coordination*: aggregation, naming, counting, etc. [3, 20, 34, 40, 76, 79]; *computability* (e.g., [7, 8, 21, 24, 28, 36, 39, 69, 75]); *control*: election, consensus, synchronization, etc. (e.g., [6, 10, 11, 15, 17, 18, 30, 36, 49, 50, 61, 68, 70, 88].

A separate area of research has been on computations by entities opportunistically moving from node to node by traversing edges when they appear. This is the classical environment of mobile agents (or robots) moving in a network, extended to dynamic networks. Also this case is one of uncontrolled generation, as the mobile agents have no control over the changes in the network; the changes are often seen as generated by the movement of *carriers.* The main research focus has been on *search* and *exploration* (e.g., [1, 2, 12, 22, 37, 41, 43, 45, 46, 58]).

4 Without Control What to Assume?

The fact that the future is unknown and under the control of an adversary means that, in order to be able to perform some meaningful computation, some *assumptions* have to be made. Usually called *a priori knowledge* and sometimes *oracles*, these assumptions restrict the universe under observation, limiting the power of the adversary.

As mentioned before, the most common assumption is that the system is *discrete synchronous*. Another common (usually hidden) assumption is that the footprint $G = (V, N)$ is finite, i.e., both N and E are finite. Let us make these assumptions. Still, they are not enough, and additional assumptions are necessary.

For example, in the non-deterministic realm, additional assumptions are made on the probability of the appearance of every edge (e.g, it obeys a Poisson process) or on the relationship beween successive snapshots $G(t)$ and $G(t + 1)$ (e.g., edge-Markovian process); e.g., see [14, 29, 31, 33]. The focus of this note is however on the *deterministic* side.

4.1 Frequency Assumptions

A type of deterministic additional assumptions are about the *frequency* of the changes. Let an edge $e \in E$ be called *transient* if it appears in a finite number of snapshots $G(t)$, *recurrent* otherwise. Notice that if there are both transient and recurrent edges, there exists a time $\tilde{t}$ after which all appearing edges are recurrent.

We say that the system $\mathcal{G}$ is *recurrent* if all edges are recurrent: $\forall e \in E, t \in, \exists t' > t : e \in E(t')$; this restriction is sometimes called *local fairness*. Example of recurrent systems are population protocols with a fair scheduler. Investigations include e.g., [1, 2, 6, 7, 8, 26, 27, 75].

A more restricted class is that of *B-bounded* recurrent systems, $B \in \mathbb{N}$, defined by the assumption $e \in E(t) \Rightarrow e \in E(t')$ where $t < t' \leq t + B$ (e.g., [1, 2, 26, 27]).

Even more restricted is the class of *P-periodic* systems, $P \in \mathbb{N}$, defined by the assumption $e \in E(t) \Rightarrow e \in E(t + P)$. Examples of periodic systems are public transports with fixed timetable, low-earth orbiting satellite (LEO) systems. Study of computing in periodic systems include [1, 2, 26, 27, 45, 46, 58, 60, 71, 72]. Notice that to determine whether or not a system is periodic is undecidable. Similiarly undecidable is to verify if a periodic system has period P. In other words, periodicity without knowledge of the period is not a useful assumption.

Note that all these frequency assumptions are restictions on the functions ψ and ρ. Another type of requirement sometimes imposed on those functions is that they should somehow reflect the behaviour of "real life" systems. With this motivation, the engineering community has developed and has been using several *mobility patterns*, i.e. restrictions on the functions ψ and ρ to mimic experimentally observed changes due to mobility of vehicles, humans, etc. (e.g., [38, 81, 87]).

4.2 Connectivity Assumptions

Of all the common assumptions we considered so far, none has any impact on the *connectivity* of the network. Indeed simultaneous end-to-end connectivity might not be guaranteed, and it is also possible that all snapshots $G(t)$ might be disconnected in spite of those assumptions. Not surprising, especially for discrete synchronous systems, a popular class of additional assumptions are those relating to *connectivity*.

The weakest such assumption is *temporal connectivity*, that is $\forall x, y \in V, t \in \mathcal{T}$ there exixts a journey $J \in \mathcal{J}_{(u,v)}$ such that $start(J) \geq t$. This assumption is typical in the engineering investigations; it is also the one used in the pioneering work of Awerbuch and Even [13].

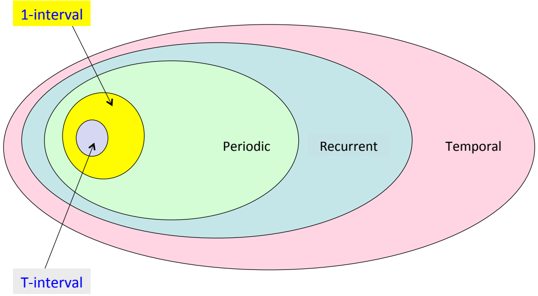


Figure 3 Connectivity assumptions.

Stronger assumptions require simultaneous end-to-end connectivity to hold at some point in time. In increasing order of requirement's strenght, we have *recurrent connectivity*: $\forall t \in \mathcal{T}, \exists\, t' \geq t:\ G(t')$ is connected (e.g., [10, 61]); *B-bounded connectivity* : $\forall t \in \mathcal{T}, \exists\, t' \leq t+B : G(t')$ is connected; and *P-periodic connectivity* : $\forall t \in \mathcal{T}, \exists\, t' \geq t : \forall j \in G(t'+jP)$ is connected.

Finally, *permanent connectivity*: $\forall t \in \mathcal{T}, G(t)$ is connected; this assumption is also known as *1-interval connectivity.* An even more stringent assumption is *permanent connectivity with persistent backbone* which requires that the same connected spanning subgraph persists for $T > 1$ consecutive snapshots: $\forall t \in \mathcal{T}, G_T(t) = \bigcap_{0 \leq j < T} G(t+j)$ is connected; this assumption is also known as *T-interval connectivity.* Both permanent and T-interval connectivity are often assumed in distributed computations (e.g., [3, 41, 43, 54, 59, 68, 69, 70, 83, 88]).

4.3 Power of Assumptions

Notice that to each set of assumptions corresponds the class of systems satisfying those assumptions. Important questions are about the *computational power* of these classes of systems. For example, is one class more powerful than another ? What is the weakest class where a given problem is solvable ? (i.e., what are the weakest assumptions which allow to solve a given problem ?)

For example, with respect to the frequency of changes, we have identified the classes $\mathcal{G}[Recurrent]$, $\mathcal{G}[Bounded]$, and $\mathcal{G}[Periodic]$. Let $\mathcal{P}[Recurrent]$, $\mathcal{P}[Bounded]$, and $\mathcal{P}[Periodic]$ be the set of problems solvable in those classes. Clearly, $\mathcal{P}[Recurrent] \subseteq \mathcal{P}[Bounded] \subseteq \mathcal{P}[Periodic]$. Consider the problem of *minimal broadcast with termination detection*: optimally diffusing some information and the source knowing (within finite time) of the completion of the process. As we discussed previously, with respect to "minimality", there are three types of journeys and thus of broadcasts: *foremost* broadcast *FoB*, in which the date of delivery is minimized at every node; *shortest* broadcast *ShB*, where the number of hops used by the broadcast is minimized relative to every node; and *fastest* broadcast *FaB*, where the overall duration of the broadcast is minimized (however late the departure be). Interestingly : $FoB \in \mathcal{P}[Recurrent]$ but $ShB \notin \mathcal{P}[Recurrent]$; furthermore $ShB \in \mathcal{P}[Bounded]$ but $FaB \notin \mathcal{P}[Bounded]$; on the other hand, $FaB \in \mathcal{P}[Periodic]$. This implies first of all a strict order on the difficulty of the three problems:

$$FoB < ShB < FaB\,.$$

It also shows that the inclusion among the set of problems is strict:

$$\mathcal{P}[Recurrent] \subsetneq \mathcal{P}[Bounded] \subsetneq \mathcal{P}[Periodic]$$

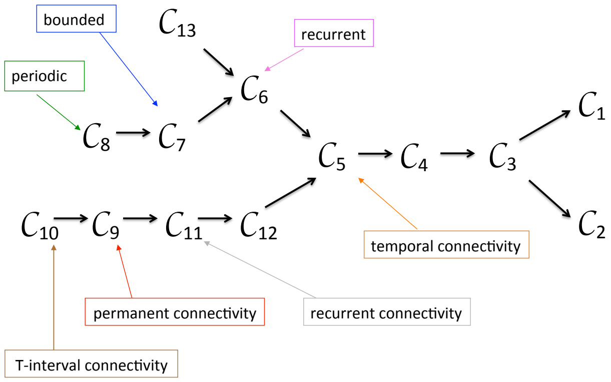


Figure 4 Hierarchy of system classes, from [28].

and thus the strict hierarchy of computational power of those graph classes:

$$\mathcal{G}[Recurrent] \prec \mathcal{G}[Bounded] \prec \mathcal{G}[Periodic] .$$

Note that these results, established in [26, 27], hold even without the discrete synchronous assumption.

It is clear that any result established under a set of assumptions gives some information about the computational impact of those assumptions. This information gives raise to a hierarchy of classes of systems, as pointed out in [28].

5 Conclusions

At the end of this note, it should be clear that researching "distributed computing in dynamic networks" means to consider a system that is (still) alive and evolving, and to understand that the computation is by necessity decentralized and localized. More precisely, the point of view is that of the computational entity, operating in an ever changing system; always in the present, the entity is only aware of the changes in which it is involved, possibly remembering the past, and in general without knowledge of the future.

The first distinction is on whether or not the computational entity has control over the topological changes in which it is involved; that is, on whether the generation is controlled or uncontrolled by the computation. In the case of controlled generation, an entity has a degree of control over the future. In the case of uncontrolled generation (a condition that occurs also in the case of mobile agents), assumptions have to be made to render computation possible. The next distinction is on what assumptions are made.

Among the open research goals the obvious ones are to establish new results and to explore new problems. An important goal is to *remove assumptions*. Indeed, when facing a problem the guiding question should be: what is the weakest assumption that makes the problem solvable?

The strongest assumption commonly made is that the system is discrete synchronous. Actually almost all the algorithmic investigations making this assumption further assume that the scheduling of the entities is *fully sysnchronous*: at each time steps, all the entities are active and participate in the distributed computation. A weaker assumption is that of a *semi-synchronous* setting: at each time step a (non-empty) subset of the entities are active, the activation choice made by a fair but adversarial scheduler. This model, quite common in the context of robotic swarms [47], has just started to be examined in the other dynamic networks contexts [41].

A larger open research direction is to remove the discrete synchronous assumption completely, and look at the more general case possible, asynchronous and continuous. As mentioned, some investigations have been carried out and results established in the general case (e.g., [26, 27, 29]). Interestingly, in some investigations that assume discrete synchronous systems, the analysis is however carried out in the continuous setting (e.g., [65]).

References

1 E. Aaron, D. Krizanc, and E. Meyerson. DMVP: Foremost waypoint coverage of time-varying graphs. In *Proc. 40th Int. Work. Graph Th. Conc. Comp. Sci. (WG)*, 29–41, 2014.

2 E. Aaron, D. Krizanc, and E. Meyerson. Multi-robot foremost coverage of time-varying graphs In *Proc. 10th ALGOSENSORS*, 22–38, 2015.

3 S. Abshoff and F. Meyer auf der Heide. Continuous aggregation in dynamic ad-hoc networks. In *Proc. 21st Int. Coll. on Structural Inf. and Comm. Compl. (SIROCCO)*, 194–209, 2014.

4 S. Abshoff, M. Benter, M. Malatyali, and F. Meyer auf der Heide. On two-party communication through dynamic networks In *Proc. 17th International Conference on Principles of Distr. Syst. (OPODIS)*, 11–22, 2013.

5 M. Ahmadi, A. Ghodselahi, F. Kuhn, and A.R. Molla. The cost of global broadcast in dynamic radio networks. In *these Proceedings (OPODIS)*, 2015.

6 D. Alistarh and R. Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proc. 42nd Int. Coll. Automata, Languages, Program. (ICALP)*, 479–491, 2015.

7 D. Angluin, J. Aspnes, Z. Diamadi, M. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

8 D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

9 M. Antony and A. Gupta. Finding a small set of high degree nodes in time-varying graphs. In *Proc. 15th IEEE Int. Symp. Wireless, Mob. Multimedia Netw. (WoWMoM)*, 1–6, 2014.

10 L. Arantes, F. Greve, P. Sens, and V. Simon. Eventual leader election in evolving mobile networks. In *Proc. of 17th Int. Conf. Principles of Distr. Syst. (OPODIS)*, 23–37, 2013.

11 J. Augustine, G. Pandurangan, and P. Robinson. Fast Byzantine agreement in dynamic networks. In *Proc. of 32nd Symp. Principles of Dist. Comp. (PODC)*, 74–83, 2013.

12 C. Avin, M. Koucky, and Z. Lotker. How to explore a fast-changing world. In *Proc. of the 35th Int. Coll. on Automata, Languages and Programming (ICALP)*, 121–132, 2008.

13 B. Awerbuch and S. Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proc. of 3rd Symp. Princip. Dist. Comp.(PODC)*, 278–281, 1984.

14 H. Baumann, P. Crescenzi, and P. Fraigniaud. Parsimonious flooding in dynamic graphs. In *Proc. of the 28th ACM Symp. on Principles of Distr. Comp. (PODC)*, 260–269, 2009.

15 A. Benchi, P. Launay, and F. Guidec. Solving consensus in opportunistic networks. In *Proc. 16th Int. Conf. on Distributed Computing and Networking (ICDCN)*, 1:1–1:10, 2015.

16 S. Bhadra and A. Ferreira. Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks. In *Proc. 2nd Intl. Conf. on Ad Hoc Networks and Wireless (ADHOC-NOW)*, 259–270, 2003.

17 M. Biely, P. Robinson, and U. Schmid. Agreement in directed dynamic networks. In *Proc. of the 19th Int. Coll. on Structural Inf. and Comm. Complexity (SIROCCO)*, 73–84, 2012.

18 M. Biely, P. Robinson, U. Schmid, M. Schwarz, and K. Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. In *Proc. of the 2nd International Conference on Networked Systems*, 2015.

19 B. Blonder, T.W. Wey, A. Dornhaus, R. James, and A. Sih. Temporal dynamics and network analysis. *Methods in Ecology and Evolution* (3):958–972, 2012.

20 Q. Bramas and S. Tixeuil. The complexity of data aggregation in static and dynamic wireless sensor networks. In *Proc. 17th Int. Symp. Stab. Safety Secur. (SSS)*, 36–50, 2015.

21 P. Brandes and F. Meyer auf der Heide. Distributed computing in fault-prone dynamic networks. In *Proc. of 4th Int. Work. Theor. Aspects Dynamic Distr. Syst.*, 9–14, 2012.

22 B. Brejova, S. Dobrev, R. Kralovic, and T. Vinar. Efficient routing in carrier-based mobile networks. *Theoretical Computer Science*, 509:113–121, 2013.

23 B. Bui-Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. of Foundations of Computer Science*, 14(2):267–285, 2003.

24 A. Casteigts, S. Chaumette, and A. Ferreira. Characterizing topological assumptions of distributed algorithms in dynamic networks. In *Proc. of 16th Int. Coll. on Structural Information and Communication Complexity (SIROCCO)*, 126–140, 2009.

25 A. Casteigts, P. Flocchini, E. Godard, N. Santoro, M. Yamashita On the expressivity of time-varying graphs. *Theoretical Computer Science*, 590:27–37, 2015

26 A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Measuring temporal lags in delay-tolerant networks. *IEEE Trans. Comp.* 63(2): 397–410, 2014.

27 A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Shortest, fastest, and foremost broadcast in dynamic networks. *Int. J. of Foundations of Comput. Sci.*, 26(4):499–522, 2015.

28 A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *Int. J. Parallel, Emergent and Distributed Syst.*, 27(5):387–408, 2012.

29 A. Chaintreau, A. Mtibaa, L. Massoulie, and C. Diot. The diameter of opportunistic mobile networks. *Communications Surveys & Tutorials*, 10(3):74–88, 2008.

30 B. Charron-Bost, M. Fugger, and T. Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP)*, 528–539, 2015.

31 A.E.F. Clementi, C. Macci, A. Monti, F. Pasquale, and R. Silvestri. Flooding time of edge-Markovian evolving graphs. *SIAM J. on Discrete Mathematics*, 24(4):1694–1712, 2010.

32 A.E.F. Clementi, A. Monti, F. Pasquale, and R. Silvestri. Information spreading in stationary Markovian evolving graphs. *IEEE Trans. Par. Distr. Syst.*, 22(9):1425–1432, 2011.

33 A.E.F. Clementi, R. Silvestri, and L. Trevisan. Information spreading in dynamic graphs. *Distributed Computing* 28(1):55–73, 2015.

34 A. Cornejo, S. Gilbert, and C. Newport. Aggregation in dynamic networks. In *Proc. Symp. Principles of Distributed Computing (PODC)*, 195–204, 2012.

35 A. Cornejo, C. Newport, S. Gollakota, J. Rao, and T.J. Giuli. Prioritized gossip in vehicular networks. *Ad Hoc Networks*, 11(1):397–409, 2013.

36 E. Coulouma and E. Godard. A characterization of dynamic networks where consensus is solvable. In *Proc. 20th Int. Coll. Structural Inf. Comm. Comp. (SIROCCO)*, 24–35, 2013.

37 O. Denysyuk and L. Rodrigues. Random walks on evolving graphs with recurring topologies. In *Proc. 28th Int. Symp. on Distributed Computing (DISC)*, 333–345, 2014.

38 A. Diab and A. Mitschele-Thiel. *Human Mobility Patterns.* IGI 2014.

39 G.A. Di Luna and R. Baldoni Non trivial computations in anonymous dynamic networks. In *these Proceedings (OPODIS)*, 2015.

40 G.A. Di Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. Conscious and unconscious counting on anonymous dynamic networks. In *Proc. 15th Int. Conf. on Dist. Comp. and Networking (ICDCN)*, 257–271, 2014.

41 G.A. Di Luna, S. Dobrev, P. Flocchini, and N. Santoro. Live exploration of dynamic rings *arXiv*: 1512.05306, 2015.

42 C.Dutta, G.Pandurangan, R.Rajaraman, Z.Sun, and E.Viola. On the complexity of information spreading in dynamic networks. In *Proc. Symp. on Disc. Alg. (SODA)* 717–736, 2013.

43 T. Erlebach, M. Hoffmann, and F. Kammer. On temporal graph exploration In *Proc. of 42nd Int. Coll. on Automata, Languages, and Programming (ICALP)*, 444–455, 2015.

44 A. Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004.

45 P. Flocchini, M. Kellett, P. Mason, and N. Santoro. Searching for black holes in subways. *Theory of Computing Systems*, 50(1):158–184, 2012.

46 P. Flocchini, B. Mans, and N. Santoro. On the exploration of time-varying networks. *Theoretical Computer Science*, 469:53–68, 2013.

47 P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots.* Morgan & Claypool, 2012.

48 E. Godard and D. Mazauric. Computing the dynamic diameter of non-deterministic dynamic networks is hard. In *Proc. 10th ALGSENSORS*, 88-102, 2014.

49 C. Gomez-Calzado, A. Lafuente, M. Larrea, and M. Raynal. Fault-tolerant leader election in mobile dynamic distributed systems. In *Proc. 19th Pacific Rim Int. Symp. on Depend. Comput. (PRDC)*, 78–87, 2013.

50 F. Greve, P. Sens, L. Arantes, and V. Simon. Eventually strong failure detector with unknown membership. *The Computer Journal*, 55(12):1507–1524, 2012.

51 B. Haeupler and F. Kuhn. Lower bounds on information dissemination in dynamic networks. In *Proc. of 26th Int. Symp. on Distributed Computing (DISC)*, 166–180, 2012.

52 F. Harary and G. Gupta. Dynamic graph models. *Math. Comp. Model.*, 25(7):79–88, 1997.

53 D. Ilcinkas, R. Klasing, and A.M. Wade. Exploration of constantly connected dynamic graphs based on cactuses. In *Proc. 21st SIROCCO*, 250–262, 2014.

54 B. Haeupler, F. Kuhn. Lower bounds on information dissemination in dynamic networks. In *Proc. 26th Int. Symp.on Distributed Computing (DISC)*, 166–180, 2012.

55 P. Holme. Network reachability of real-world contact sequences. *Physical Review E*, 71(4):46119, 2005.

56 P. Holme. Modern temporal network theory: A colloquium. *Eur. Phys. J. B*, 88: 234, 2015.

57 P. Holme and J. Saramaki. Temporal networks. *Physics Reports*, 519:97–125, 2012.

58 D. Ilcinkas and A.M. Wade. On the power of waiting when exploring public transportation systems. *Proc. 15th Int. Conf. on Principles of Dist. Syst. (OPODIS)*, 451–464, 2011.

59 D. Ilcinkas and A.M. Wade. Exploration of the T-Interval-connected dynamic graphs: the case of the ring. In *Proceedings 20th SIROCCO*, 13–23, 2013.

60 R. Jathar, V. Yadav, and A. Gupta. Using periodic contacts for efficient routing in delay tolerant networks. *Ad Hoc & Sensor Wireless Networks*, 22(1,2):283–308, 2014.

61 D. Jeanneau, T. Rieutord, L. Arantes, and P. Sens A failure detector for k-set agreement in asynchronous dynamic systems. INRIA Research Report 8727, 2015.

62 H. Kim, R. Anderson. Temporal node centrality in complex networks. *Phys. Rev. E*, 85:1–8, 2012

63 M. Konschake, H.H.K. Lentz, F.J. Conraths, P. Hovel, and T. Selhorst. On the robustness of in-and out-components in a temporal network. *PloS One*, 8(2):e55223, 2013.

64 G. Kossinets, J. Kleinberg, and D. Watts. The structure of information pathways in a social communication network. In *Proceedings of the 14th Int. Conference on Knowledge Discovery and Data Mining (KDD)*, 435–443, 2008.

65 V. Kostakos. Temporal graphs. *Physica A*, 388(6): 1007–1023, 2009.

66 L. Kovanen, M. Karsai, K. Kaski, J. Kertesz, J. Saramaki. Temporal motifs in time-dependent networks. *J. Stat. Mech. Theor. Exp.*, 2011(11): 11005, 2011.

67 M. Krivela, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter. Multilayer networks. *J. Complex Networks*, 2(3): 203–271, 2014.

68 F. Kuhn, T. Locher, and R. Oshman. Gradient clock synchronization in dynamic networks *Theory of Computing Systems*, 49(4):781–816, 2011.

69 F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proc. of the 42nd ACM Symp. on Theory of Computing (STOC)*, 513–522, 2010.

70 F. Kuhn, Y. Moses, and R. Oshman. Coordinated consensus in dynamic networks. In *Proc. 30th Symp. on Principles of Distributed Computing (PODC)*, 1–10, 2011.

71 C. Liu and J. Wu. Scalable routing in cyclic mobile networks. *IEEE Transactions on Parallel and Distributed Systems*, 20(9): 1325–1338, 2009.

72 C. Mergenci and I. Korpeoglu. Routing in delay tolerant networks with periodic connections. *EURASIP J. Wireless Communications and Networking*, 202, 2015.

73 G. B. Mertzios, O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Temporal network optimization subject to connectivity constraints. In *Proc. 40th International Colloquium on Automata, Languages, and Programming (ICALP)*, 657–668, 2013.

74 O. Michail. An introduction to temporal graphs: An algorithmic perspective. In *Algorithms, Probability, Networks, and Games*, Springer, 308-343, 2015.

75 O. Michail, I. Chatzigiannakis, and P.G.. Spirakis. Mediated population protocols. *Theor. Comput. Sci.*, 412(22):2434–2450, 2011.

76 O.Michail, I.Chatzigiannakis, and P.G.Spirakis. Naming and counting in anonymous unknown dynamic networks In *Proc. 15th Int. Symp. Stab., Safety, Sec. (SSS)*, 281–295, 2013.

77 O. Michail, I. Chatzigiannakis, and P.G. Spirakis. Causality, influence, and computation in possibly disconnected synchronous dynamic networks. *Journal of Parallel and Distributed Computing*, 74(1):2016–2026, 2014.

78 O. Michail and P.G. Spirakis. Traveling salesman problems in temporal graphs. In *Proc. 39th Int. Symp. on Mathematical Foundations of Computer Science (MFCS)*, 553-564, 2014.

79 A. Milani and M.A. Mosteiro. A faster counting algorithm for anonymous dynamic networks. In *these Proceedings (OPODIS)*, 2015.

80 A. Mtibaa, A. Chaintreau, L. Massoulie, and C.Diot. The diameter of opportunistic mobile networks. In *Proc 3rd Int. Conf. Emerging Networking Exp. Technol. (CoNEXT)*, 12, 2007

81 M. Musolesi and C. Mascolo. A community based mobility model for ad hoc network research. In *Proc 2nd Int. Work. Multi-Hop Ad Hoc Networks*, 31–38, 2006.

82 V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, V. Latora. Components in time-varying graphs. *Chaos*, 22(2):023101, 2012.

83 R. O'Dell and R. Wattenhofer. Information dissemination in highly dynamic graphs. In *Proc. of the Joint Workshop on Foundations of Mobile Computing*, 104–110, 2005.

84 Y. Pan and X. Li. Structural controllability and controlling centrality of temporal networks. *PLOS ONE*, 9(4): e94998, 2014.

85 U. Redmond, M. Harrigan, and P. Cunningham. Identifying time-respecting subgraphs in temporal networks. In *Proceedings Europ. Conf. on Machine Learning*. 2012

86 F.J. Ros and P.M. Ruiz. Minimum broadcasting structure for optimal data dissemination in vehicular networks. *IEEE Transactions on Vehicular Technology*, 62(8):3964–3973, 2013.

87 A.K. Saha and D. Johnson. Modeling mobility for vehicular ad-hoc networks. In *Proceedings 1st ACM Int. Workshop on Vehicular Ad Hoc Networks (VANET)*, 2004

88 G. Sharma and C. Bush. Distributed queueing in dynamic networks. *Parallel Processing Letters*, 25(2), 2015.

89 T.A.B. Snijders. The statistical evaluation of social network dynamics. In *Sociological Methodology* (M.E. Sobel and M.P. Becker Eds), Blackwell, 361–395, 2001.

90 J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora. Small-world behavior in time-varying graphs. *Physical Review E*, 81(5):055101, 2010.

91 J. Whitbeck, M. Dias de Amorim, V. Conan, and J.-L. Guillaume. Temporal reachability graphs. In *Proceedings 8th International Conference on Mobile Computing and Networking (MOBICOM)*, 377–388, 2012.

92 M. Wildemann, M. Rudolf, and M. Paradies. The time has come: Traversal and reachability in time-varying graphs. In *Proc. VLDB Workshop on Big-Graphs Online Querying*, 2015.

93 Z. Yang, S. Yat-sen, W. Wu, Y. Chen, and J. Zhang. Efficient information dissemination in dynamic networks. In *Proc. 42nd Int. Conf. on Parallel Proces. (ICPP)*, 603–610, 2013.

94 Z. Zhang. Routing in intermittently connected mobile ad hoc networks and delay tolerant networks: Overview and challenges. *IEEE Communications Surveys & Tutorials*, 8(1):24–37, 2006.

Space Bounds for Reliable Storage: Fundamental Limits of Coding

Alexander Spiegelman[1], Yuval Cassuto[2], Gregory Chockler[3], and Idit Keidar[4]

1 Dept. of Electrical Engineering, Technion, Haifa, Israel
sashas@tx.technion.ac.il

2 Dept. of Electrical Engineering, Technion, Haifa, Israel
ycassuto@ee.technion.ac.il

3 CS Department, Royal Holloway, London, UK
gregory.chockler@rhul.ac.uk

4 Dept. of Electrical Engineering, Technion, Haifa, Israel
idish@ee.technion.ac.il

Abstract

We present here a synopsis of a keynote presentation given by Idit Keidar at *OPODIS 2015*, the International Conference on Principles of Distributed Systems, which took place in Rennes, France, on December 14-17 2015. More details may be found in [9].

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases distributed storage, impossibility

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.4

Category Keynote

1 Introduction

In recent years we see an exponential increase in storage capacity demands, creating a need for *big data storage* solutions. Additionally, today's economy emphasizes consolidation, giving rise to massive data centers and clouds. In this era, distributed storage plays a key role. Data is typically stored on a collection of *storage nodes* and is accessed by clients over the network. Due to the geographical spread of such systems, communication is usually modeled as *asynchronous*.

Given the inherent failure-prone nature of storage and network components, a reliable distributed storage algorithm must store redundant information in order to allow data to remain available when storage nodes fail or go offline. The most common approach to achieve this is via *replication* [2], i.e., storing copies of each data block on multiple nodes. In asynchronous settings, $2f+1$ replicas are needed in order to tolerate f failures [2]. Given the immense size of data, the storage cost of replication is significant. Some previous works have attempted to mitigate this cost via the use of erasure codes [1, 3, 6, 4, 10, 5].

Indeed, code-based solutions can reduce the storage cost as long as data is not accessed concurrently by multiple clients. For example, if the data size is D bits and a single failure needs to be tolerated, erasure-coded storage ideally requires $(k+2)D/k$ bits for some parameter $k > 1$ instead of the $3D$ bits needed for replication. But as concurrency grows, the cost of erasure-coded storage grows with it: when c clients access the storage concurrently, existing code-based algorithms store $O(cD)$ bits. Intuitively, this occurs because coded data

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 4; pp. 4:1–4:3
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

cannot be reconstructed from a single storage node. Therefore, writing coded data requires coordination – old data cannot be deleted before ensuring that sufficiently many blocks of the new data are in place. This is in contrast with replication, where data can always be read coherently from a single copy, and so old data may be safely overwritten without coordination.

In this work we prove that this extra cost is inherent, by showing a bound on the storage complexity of asynchronous reliable distributed storage algorithms. Our bound takes into account three problem parameters: f, c, and D, where f is the number of storage node failures tolerated (client failures are unrestricted), c is the concurrency allowed by the algorithm, and D is the data size. For these parameters, we prove that the storage complexity is $\Theta(D \cdot min(f, c))$. Asymptotically, this means either a storage cost as high as that of replication, or as high as keeping as many versions of the data as the concurrency level.

2 Lower bound

Our formal results are proven for emulations of a lock-free multi-reader multi-writer *regular register* [7, 8].

For our lower bound, we consider algorithms that use (arbitrary) black-box encoding schemes, which produce coded blocks of a given stored value independently of other values. We assume that the storage consists of such coded blocks, in addition to possibly unbounded data-independent meta-data, which we neglect. Given our storage model, every data bit in the storage can be associated with a unique written value. Therefore, we measure the storage cost of every value as the total number of bits in the storage that are associated with this value; the total storage cost as the sum of the costs of all values.

We define a parameter $0 \leq \ell \leq D$, and observe that if a value v is associated with fewer than $D - \ell$ bits in the storage, then more than ℓ bits (associated with v) still needed to be written to the storage before v can be read.

Note that we use here a fundamental information-theoretic "pigeonhole" argument that any representation, either coded or un-coded, cannot guarantee to recover a value $v \in \mathbb{V}$ precisely from fewer than $D = \log_2 |\mathbb{V}|$ bits. This argument excludes common storage-reduction techniques like compression and de-duplication, which only work in probabilistic setups and with assumptions on the written data.

Given the above observation, we define a particular adversary behavior and prove that it drives the storage to a state where either (1) $f + 1$ storage nodes hold at least $\ell + 1$ bits each, or (2) the storage holds at least $D - \ell$ bits of c different values. Now, picking $\ell = D/2$ implies our lower bound on storage cost:

▶ **Theorem 1.** *The storage cost of any algorithm that simulates a regular lock-free register with up to f storage node failures, c concurrent writes, and a value domain of 2^D bits is $\Omega(D \cdot min(f, c))$ bits.*

3 Algorithm

Finally, we present an adaptive reliable storage algorithm whose storage cost is $O(D{\cdot}min(f, c))$. We achieve this by combining the advantages of replication and erasure coding. Our algorithm does not assume any a priori bound on concurrency; rather, it uses erasure codes when concurrency is low and switches to replication when it is high.

References

1 Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 336–345. IEEE, 2005.

2 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

3 Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 115–124. IEEE, 2006.

4 Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 253–260. IEEE, 2014.

5 Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic erasure-coded distributed storage. In *Proceedings of the 22nd International Symposium on Distributed Computing*, DISC'08, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:10.1007/978-3-540-87779-0_13`.

6 Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Dependable Systems and Networks, 2004 International Conference on*, pages 135–144. IEEE, 2004.

7 Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986.

8 Cheng Shao, Jennifer L Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011.

9 Alexander Spiegelman, Yuval Cassuto, Gregory Chockler, and Idit Keidar. Space bounds for reliable storage: Fundamental limits of coding. *arXiv preprint arXiv:1507.05169*, 2015.

10 Zhiying Wang and Viveck Cadambe. Multi-version coding in distributed storage. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, pages 871–875. IEEE, 2014.

Blockchain-Based Consensus*

Juan A. Garay

Yahoo Labs, Sunnyvale, USA
`garay@yahoo-inc.com`

Abstract

Distributed consensus (aka *Byzantine agreement* [Pease, Shostak & Lamport, 1980]) is one of the fundamental problems in fault-tolerant distributed computing and cryptographic protocols. It requires correct participants (parties) to reach agreement on initially held values despite the arbitrary behavior of some of them, with the additional requirement (known as *Validity*) that if all the correct participants start off with the same value, then that must be the decision value. The problem has been studied extensively in both the unconditional setting (where no assumptions are made about the computational power of the adversary) and the cryptographic setting, and efficient (i.e., polynomial-time) solutions exist tolerating the optimal number of misbehaving parties and running in the optimal number of rounds, on networks with pairwise authenticated channels.

In many interesting scenarios, however, such as "peer-to-peer" networks, where parties come and go as they please and there are no prior relations among them, such infrastructure (pairwise authenticated channels, public-key infrastructure) is unavailable, thus raising the question whether anything "interesting" can be achieved. In this talk we answer this question in the affirmative, presenting two new probabilistic consensus protocols based on "proofs of work" (POWs, aka "moderately hard functions," "cryptographic puzzles" [Dwork & Naor, 1992]), the technology underlying Bitcoin, the first and most popular decentralized cryptocurrency to date. (In Bitcoin, POWs are implemented using the SHA-256 cryptographic hash function, by finding preimages that produce values in a given smaller domain.)

In more detail, we first extract and analyze the core of the Bitcoin protocol, which we term the Bitcoin *backbone*, and prove two fundamental properties of its "blockchain" approach which we call "common prefix" and "chain quality." The consensus protocols can then be built as applications on top of the backbone protocol, with the Agreement and Validity properties following from common prefix and chain quality, respectively. The first protocol works assuming the adversary's hashing power is bounded by $\frac{1}{3}$ of the network's total hashing power. The second consensus protocol is more elaborate, relies on the notion of robust transaction ledgers, which capture the essence of Bitcoin's operation as a cryptocurrency, and works assuming the adversary's hashing power is strictly less than $\frac{1}{2}$.

1998 ACM Subject Classification C.2.4 Distributed Systems, D.4.6 Security and Protection

Keywords and phrases Distributed consensus; cryptocurrencies; cryptographic protocols

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.5

Category Keynote

[*] This invited talk is based on "The Bitcoin Backbone Protocol: Analysis and Applications," appearing in *Proc. Eurocrypt 2015*, joint work with Aggelos Kiayias and Nikos Leonardos.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 5; pp. 5:1–5:1
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Approximation of Distances and Shortest Paths in the Broadcast Congest Clique*

Stephan Holzer[1] and Nathan Pinsker[2]

1 **Massachusetts Institute of Technology (MIT), Cambridge, USA**
`holzer@mit.edu`

1 **Massachusetts Institute of Technology (MIT), Cambridge, USA**
`npinsker@mit.edu`

Abstract

We study the broadcast version of the CONGEST-CLIQUE model of distributed computing. This model operates in synchronized rounds; in each round, any node in a network of size n can send the same message (i.e. broadcast a message) of limited size to every other node in the network. Nanongkai presented in [STOC'14 [25]] a randomized $(2 + o(1))$-approximation algorithm to compute all pairs shortest paths (APSP) in time[1] $\tilde{\mathcal{O}}(\sqrt{n})$ on weighted graphs. We complement this result by proving that any randomized $(2 - o(1))$-approximation of APSP and $(2 - o(1))$-approximation of the diameter of a graph takes $\tilde{\Omega}(n)$ time in the worst case. This demonstrates that getting a negligible improvement in the approximation factor requires significantly more time. Furthermore this bound implies that already computing a $(2 - o(1))$-approximation of all pairs shortest paths is among the hardest graph-problems in the broadcast-version of the CONGEST-CLIQUE model, as any graph-problem where each node receives a linear amount of input can be solved trivially in linear time in this model. This contrasts a recent $(1 + o(1))$-approximation for APSP that runs in time $\mathcal{O}(n^{0.15715})$ and an exact algorithm for APSP that runs in time $\tilde{\mathcal{O}}(n^{1/3})$ in the unicast version of the CONGEST-CLIQUE model, a more powerful variant of the broadcast version.

This lower bound in the broadcast CONGEST-CLIQUE model is derived by first establishing a new lower bound for $(2 - o(1))$-approximating the diameter in weighted graphs in the CONGEST model, which is of independent interest. This lower bound is then transferred to the CONGEST-CLIQUE model.

On the positive side we provide a deterministic version of Nanongkai's $(2+o(1))$-approximation algorithm for APSP [25]. To do so we present a fast deterministic construction of small hitting sets. We also show how to replace another randomized part within Nanongkai's algorithm with a deterministic source-detection algorithm designed for the CONGEST model in [21].

1998 ACM Subject Classification C.2.1 Distributed networks

Keywords and phrases distributed computing, distributed algorithms, approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.6

1 Introduction

In a distributed message passing model a network is classically represented as a graph. In this graph any node can send (pass) one message to its neighbors in every round. There are two major research directions concerning message passing models.

* Work supported by the following grants: AFOSR Contract Number FA9550-13-1-0042, NSF Award 0939370-CCF, NSF Award CCF-1217506, NSF Award number CCF-AF-0937274.

[1] We use the convention that $\tilde{\Omega}(f(n))$ is essentially $\Omega(f(n)/\text{polylog} f(n))$ and $\tilde{\mathcal{O}}(f(n))$ is essentially $\mathcal{O}(f(n)\text{polylog} f(n))$.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 6; pp. 6:1–6:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The first research direction deals with determining the locality and congestion of problems. Using the LOCAL model [29], where message-size is unbounded, one tries to characterize the locality of problems, which is the ability of a node to make decisions regarding a problem purely based on information on its local neighborhood in a graph. Using the CONGEST model [29], one tries to characterize the delays caused by congestion. We assume for this model that message sizes are bounded to $\mathcal{O}(\log n)$ bits per round, and weights are encodable in $\mathcal{O}(\log n)$ bits. Congestion arises due to bottlenecks in the network that do not provide enough bandwidth during the computation. Both, the LOCAL model and the CONGEST model are classic models that have received a great deal of attention in the past decades. Recently it was pointed out in [28] that the CONGEST model does not avoid interference from locality issues, while the LOCAL model avoids interference from congestion. To be more precise, congestion is completely avoided in the LOCAL model due to unlimited bandwidth. On the other hand the complexity of algorithms in the CONGEST model may still depend on the local structure of a graph (e.g. lower bounds transfer from the LOCAL model). To truly separate the study of congestion from locality, one needs to consider networks that avoid locality issues. These are e.g. networks in which each node is directly connected to any other node in the network (represented by a clique), which is a network in which any graph problem can be solved within one round in case unlimited bandwidth is available. Such a model was introduced earlier by Lotker et al. [23] with the intention to study overlay networks that have this property and was coined the CONGEST-CLIQUE model. Examples of parallel systems design that recently provided additional motivation to this original motivation to study the CONGEST-CLIQUE as an overlay network [23] are included in Section 2. Note that in the broadcast setting, a simple algorithm can solve the vast majority of problems in linear time: each node v can simply broadcast the IDs of all of v's neighbors and weights of incident edges in time $\mathcal{O}(n)$. Then each node in the network has full information on the graph and can perform any computation (including e.g. NP-complete problems) internally, which does not contribute to the runtime.

The second research direction focuses on determining the power of broadcast compared to (multi-)unicast. Broadcast denotes the setting in which a node can only send the same message to all its neighbors at the same time, while in a (multi-)unicast setting each node can send different messages to different neighbors at the same time.

Results of this paper push both research directions. To be more precise, we present a linear lower bound and new improved bounds for a broadcast model (the BCC model, see definition below) that purely studies congestion.

▶ **Definition 1.** When applied to the CONGEST-CLIQUE model, we denote by UCC model the (multiple-)unicast version of the CONGEST-CLIQUE model, and by BCC model the broadcast version of the CONGEST-CLIQUE model [9, 23].

1.1 Contribution

In this context, this paper extends the work of [9, 15, 25]. Drucker, Kuhn and Oshman [9] started to study the difference in computational power between the UCC and the BCC models. Like [9] we present a linear lower bound in the BCC model. The lower bounds of [9] were the first deterministic (and conditional randomized) linear lower bounds in this model. Their lower bounds consider subgraph detection. Ours are the first unconditional randomized linear lower bounds, while we consider $(2 - o(1))$-approximations of APSP and diameter. Three main conclusions from this result are:

Table 1 Summary of new and previous results for problems we study on positively weighted graphs in the BCC model. Recent results of [4] in the UCC model are summarized in Section 2.

approx. factor	APSP	Diameter	SSSP
1	$\mathcal{O}(n)^{\#}$	$\mathcal{O}(n)^{\#}$	$\widetilde{\mathcal{O}}(\sqrt{n})^{*}$
$2 - o(1)$	$\widetilde{\mathcal{O}}(n)^{\#}, \widetilde{\Omega}(n)^{\dagger}$	$\widetilde{\mathcal{O}}(n)^{\#}, \widetilde{\Omega}(n)^{\dagger}$	—
2	—	$\widetilde{\mathcal{O}}(\sqrt{n})^{*}$	—
$2 + o(1)$	$\widetilde{\mathcal{O}}(\sqrt{n})^{\ddagger}$	$\widetilde{\mathcal{O}}(\sqrt{n})^{\ddagger}$	—

#) Trivial bound: collect the whole topology in a single node, perform computation internally.
∗) Nanongkai's SSSP algorithm [25]. See Remark 14 for the diameter approximation.
†) Our randomized lower bound, see Theorem 12.
‡) Our deterministic version of the randomized algorithm of [25], see Theorem 19.

- There is a (at least quadratic) difference in the complexity between computing a $(2+o(1))$-approximation [25] and a $(2-o(1))$-approximation of APSP in this model.
- Computing a $(2-o(1))$-approximation of APSP is among the hardest graph-problems in the BCC model, as any graph-problem (with $\mathcal{O}(\log n)$-encodable weights) can be solved in linear time.
- There is a clear separation between the UCC and BCC model with respect to APSP computation. Our lower bounds contrast the results of [4], who showed that e.g. even exact APSP can be solved in the UCC model within $\tilde{\mathcal{O}}(n^{1/3})$ time and $(1+o(1))$-approximated in time $\mathcal{O}(n^{0.15715})$.

Note that this lower bound strengthens the $\tilde{\Omega}(\sqrt{n})$ lower bound for exact computation of APSP in the BCC model by [4] in terms of runtime and extends it to approximations.

Technical Overview: To obtain our lower bounds, we first use techniques of Frischknecht et al. [10] to derive an $\tilde{\Omega}(n)$-round lower bound to $(2-o(1))$-approximate the diameter of weighted graphs in the CONGEST model. This implies an $\tilde{\Omega}(n)$-round lower bound to $(2-o(1))$-approximate APSP. To prove our lower bounds, we modify a construction for unweighted graphs that was claimed in [14] and can also be found in [32] to the weighted setting. We then use this construction to transfer lower bounds for set disjointness from two-party communication complexity (first studied by Kushilevitz [20]). Next we transfer this lower bound from the CONGEST model to the BCC model.

Apart from these lower bounds, we derive positive results on computing APSP by extending the line of work of [15, 25]. We start by replacing the randomized parts of a recent result by Nanongkai [25], who presented an algorithm for a $(2+o(1))$-approximation of the all-pairs shortest paths problem in the BCC model in $\tilde{\mathcal{O}}(\sqrt{n})$ rounds, with deterministic ones. We then show that the resulting algorithm can be transferred to the UCC model with an improvement in runtime.

1.2 Structure of the Paper

We review related work in Section 2 and define the computation models and terminology that we work with in Section 3. Our lower bounds are presented in Section 4, where present a review of two-party communication complexity, state the lower bounds for the CONGEST model and transfer them to the BCC model. A key-ingredient for our upper bounds is a deterministic hitting set construction, which we present in Section 5. Finally, in Section 6, we present our deterministic version of Nanongkai's all-pairs shortest paths approximation algorithm in the BCC model. We conclude by briefly mentioning some open problems and directions for future work in Section 7.

2 Related Work

Algorithms in the BCC and UCC models: The first to study the CONGEST-CLIQUE model were Lotker et al. [23], where they presented an $\mathcal{O}(\log\log n)$-round algorithm for constructing a minimum spanning tree in the UCC model. This was improved by Pemmaraju and Sardeshmukh to $\mathcal{O}(\log\log\log n)$ in [30]. Lenzen obtained in [22] an $\mathcal{O}(1)$-round algorithm in the UCC model for simultaneously routing n messages per vertex to their assigned destination nodes, as well as an $\mathcal{O}(1)$ algorithm for sorting $\mathcal{O}(n^2)$ numbers, given that each vertex begins the algorithm knowing $\mathcal{O}(n)$ numbers. Independently Patt-Shamir and Teplitsky [28] showed a similar, but slightly weaker result on sorting in the UCC model. Later Hegeman et al. [13] provided constant and near-constant (expected) time algorithms for problems such as computing a 3-ruling set, a constant-approximation to metric facility location, and (under some assumptions) a constant-factor approximations to the minimum spanning tree in the UCC model. Holzer [14] provided a deterministic $\mathcal{O}(\sqrt{n})$-algorithm for exact unweighted SSSP (equivalent to computing a breadth first search tree) in the BCC model. Independently Nanongkai [25] provided randomized (w.h.p.) algorithms in the BCC model that take $\tilde{\mathcal{O}}(n^{1/2})$ rounds to compute (exact) SSSP, and $\tilde{\mathcal{O}}(n^{1/2})$ rounds to $(2+o(1))$-approximate APSP on weighted graphs. Much of our work for deterministic APSP builds off [25], primarily on his idea of "shortcut edges", which do not change the weighted shortest path length between any two nodes but decrease the diameter of the graph. This is combined with a deterministic h-hop multi-source shortest paths scheduling technique implied by the source-detection algorithm of Lenzen and Peleg [21], which works in the broadcast version of the CONGEST model. Note that other versions that could have been used, such as the one presented in [7, 14], only work in the (multi-)unicast version. Recently Censor-Hillel, Kaski, et al. [4] transferred fast matrix multiplication algorithms into the UCC model using results from [22] and derived a runtime of $\mathcal{O}(n^{1/3})$ in semirings and $\mathcal{O}(n^{0.15715})$ in rings. Using this they obtain an $\mathcal{O}(n^{0.15715})$ algorithm for triangle detection and undirected unweighted APSP. Both papers also solve APSP on directed weighted graphs in time $\tilde{\mathcal{O}}(n^{1/3})$. In addition [4] presents an $(1+o(1))$-approximation for exact directed weighted APSP in time $\mathcal{O}(n^{0.15715})$, while [4] derives results for fast diameter and girth computation as well as for 4-cycle detection.

Lower bounds in the BCC and UCC models: Drucker et al. [9] were the first to provide lower bounds in the BCC model. They derived these bounds by transferring lower bounds for set disjointness in the 3-party NOF model to the congested clique. In addition [9] showed that "a slightly super-constant lower bound on the number of rounds required to compute some explicit function in the unicast CONGEST-CLIQUE model (when message size is 1) would imply a new lower bound on ACC (constant depth circuits), and an $\Omega(\log\log n)$ lower bound for the unicast CONGEST-CLIQUE model would imply new a lower bound for threshold circuits (the class TC). While explicit lower bounds in the UCC model remain open and might have a major impact to other fields of (Theoretical) Computer Science as mentioned above, they argue nonconstructively that most problems have a linear lower bound in the UCC model with a counting argument. Independent and simultaneously to us, the authors of [4] presented an $\tilde{\Omega}(\sqrt{n})$ lower bound for APSP in the BCC model, which they derive from matrix multiplication lower bounds that they state.

Lower bounds in the CONGEST model (all – including ours – in the unicast version): Frischknecht et al. [10] (which is based on [5]) showed an $\tilde{\Omega}(n)$ lower bound for exact computation of the diameter of an unweighted graph. In this paper we draw on these ideas to obtain lower bounds for (2-o(1))-approximating the diameter of weighted graphs in the BCC model. Note that also Nanongkai [25] presents an $\tilde{\Omega}(n)$-time lower bound for any $poly(n)$-approximation algorithm for APSP on weighted graphs in the CONGEST model and shows that any $\alpha(n)$-approximation of APSP on unweighted graphs requires $\tilde{\Omega}(n/\alpha(n))$ time. However, his proof relies on an information-theoretic argument and uses a star-shaped graph such that it cannot be extended to the BCC model, as in this model every node could simply broadcast its distance from the center to all other nodes. Assuming a girth conjecture, Izumi and Wattenhofer show in [17] that constructing distance oracles with stretch $2t$ in unweighted (weighted) graphs takes $\Omega(n^{1/(t+1)})$ rounds ($\Omega(n^{\frac{1}{2}+\frac{5}{t}})$ rounds). When $o(n^{\epsilon})$ label size is required, assuming the girth-conjecture can be dropped. In contrast to this, our lower bound does not assume relabeling. Our construction and the construction of [17] build on top of [10] and appeared at the same time: [17] and the technical report [16].

Connections to systems and other models: Finally we want to provide examples of parallel systems that might benefit from theoretical results in the CONGEST-CLIQUE model. These include systems that provide all-to-all communication between 10,000 nodes at full bandwidth [27]. In addition [12] showed a close connection between the UCC model and popular parallel systems such as MapReduce [6] and analyzed which kind of algorithms for the UCC model can be simulated directly in MapReduce. Pregel [24] is a system that simulates algorithms designed for message-passing models such as the CONGEST model (the input graph is split among several machines). Klauk et al. [19] study large-scale graph processing systems such as Pregel [24] in a theoretic way (k-machine model), which also includes the CONGEST-CLIQUE model. Finally, the authors of [9] pointed out that the BCC model is used in streaming [1], cryptology [11] and mechanism design [8]. They also establish connections between the UCC and ACC as well as TC0 circuits.

3 Model and Definitions

We first introduce the CONGEST model and then derive the CONGEST-CLIQUE model, which is at the center of this paper.

The CONGEST Model: Our network is represented by an undirected graph $G = (V, E)$, where nodes V model processors or computers and edges E model links between the processors. Edges can have associated *weights* $w : E \to \{a/p \mid a \in \{1, \ldots, p^2\} \subset \mathbb{N}\}$ for some $p \in poly(n)$. This ensures that each weight is a positive multiple of $1/p$ and can be encoded in $\mathcal{O}(\log n)$ bits. Two nodes can communicate directly with each other if and only if they are connected by some edge from set E. We also assume that the nodes have unique IDs in the range of $\{1, \ldots, poly(n)\}$ and infinite computational power.[2] At the beginning, each node knows only the IDs of its neighbors and the weights of its incident edges.

[2] This assumption is made by the model because it is used to study communication complexity. Note that we do not make use of this, as our algorithms perform efficient computations.

We consider a model where nodes can send messages to their neighbors over synchronous rounds of communication. During a round, each node u can send a message of B bits through each edge connecting u to some other vertex v. We assume $B = \mathcal{O}(\log n)$ during our algorithms, which is the standard choice [29] and state our lower bounds depending on arbitrary B. The message will arrive at node v at the end of the round. We analyze the performance of an algorithm in this model by measuring the worst-case number of communication rounds required for the algorithm to complete.

Let $\mathcal{A}$ be the set of distributed deterministic algorithms that evaluate a function g on an underlying graph $G \in \mathbb{G}_n$ over n nodes, where $\mathbb{G}_n$ is the set of connected graphs over these nodes. We define the *distrubuted round complexity* of an integer-valued function g as follows:

▶ **Definition 2** (Distributed Round Complexity)**.** The distributed round complexity $R^{dc}(g)$ is defined to be $\min_{A \in \mathcal{A}} \max_{G \in \mathbb{G}_n} R^{dc}(A(G))$. In other words, $R^{dc}(A(G))$ represents the number of rounds that an algorithm $A \in \mathcal{A}$ needs in order to compute $g(G)$.

We denote by $R^{dc-pub}_{\varepsilon}(g)$ the (public coin[3]) randomized round complexity of g when the algorithms have access to public coin randomness and compute the desired output with an error probability smaller than ε.

The CONGEST-CLIQUE Model: In this model every vertex in a network G can directly communicate with every other vertex in G. Note that although the communication graph is a clique, we are interested in solving a problem on a subgraph G of the clique. Working under the broadcast and (multi-)unicast versions of the CONGEST model while making this assumption gives us the BCC model and the UCC model, respectively.

Problems and Definitions: For any nodes u and $v \in V$, a *(u,v)-path* P is a path $(u = x_0, x_1, \ldots, x_l = v)$ where $(x_i, x_{i+1}) \in E$ for all i. We define the weight of a path P to be $w(P) := \sum_{i=0}^{l-1} w(x_i, x_{i+1})$. Let $P_G(u,v)$ denote the set of all (u,v)-paths in G. We define $d_w(u,v) = \min_{P \in P_G(u,v)} w(P)$; in other words, $d_w(u,v)$ is the weight of the shortest (weighted) path from u to v in G. The (weighted) *diameter* D_w of (G,w) is defined as $\max_{u,v \in V} \ d_w(u,v)$. For unweighted graphs G (i.e. $w(e) = 1$ for all $e \in E$), we omit w from our notations. In particular, $d(u,v)$ is the (hop-)distance between u and v in G, and D is the diameter of the unweighted network G.

▶ **Definition 3** (Single Source Shortest Paths and All-Pairs Shortest Paths)**.** In the (weighted) single source shortest paths problem (SSSP), we are given a weighted network (G,w) and a source node s. We want each node v to know the distance $d_w(s,v)$ between itself and s. In the (weighted) all pairs shortest paths problem (APSP), each node $v \in V$ needs to know $d_w(u,v)$ for all $u \in V$.

For any α, we say an algorithm A is an α-*approximation* algorithm for SSSP if each node v obtains a value $\widetilde{d}_w(s,v)$ from A, such that $d_w(s,v) \leq \widetilde{d}_w(s,v) \leq \alpha \cdot d_w(s,v)$. Similarly, we say A is an α-approximation algorithm for APSP if each node v obtains values $\widetilde{d}(u,v)$ such that $d_w(u,v) \leq \widetilde{d}_w(u,v) \leq \alpha d_w(u,v)$ for all u. Note that this is one-sided error; an algorithm A is not an α-approximation algorithm if it ever outputs a value $\widetilde{d}_w(s,v) < d_w(s,v)$.

[3] This is mainly of interest for our lower bounds. Our algorithms also work with private randomness.

4 Lower Bounds for Weighted and Unweighted Diameter Computation and Approximation

Frischknecht et al. proved in [10] that any algorithm that computes the exact diameter of an unweighted graph requires at least $\Omega(\frac{n}{B})$ rounds of communication in the unicast CONGEST model. Note that they consider an arbitrary message-size B; the CONGEST model typically considers only $B = \mathcal{O}(\log n)$. We consider arbitrary B here as well. Their lower bound is achieved by constructing a reduction from the two-party communication problem of *set disjointness* to the problem of calculating the diameter of a particular unweighted graph G. We extend their construction that considers exact computation of the diameter of an unweighted graphs to the case of $(2 - 1/poly(n))$-approximating the diameter in a (positively) weighted graph. This is done by assigning weights to the edges in their (unweighted) construction in a convenient way and deriving the approximation-factor. We start by reviewing basic tools from two-party communication complexity and then present the modification of the construction of [10] for the CONGEST model in Section 4.3. Subsequently we transfer this bound to the BCC model.

4.1 A Review of Basic Two-Party Communication Complexity

It is necessary to review the basics of two-party communication complexity in order to present our results in a self-contained way. In the remaining part of this subsection we restate the presentation given in [15] only for completeness and convenience of the reader.

Two computationally unbounded parties Alice and Bob each receive a k-bit string $a \in \{0,1\}^k$ and $b \in \{0,1\}^k$ respectively. Alice and Bob can communicate with each other one bit at a time and want to evaluate a function $h : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}$ on their input. We assume that Alice and Bob have access to public randomness for their computation and we are interested in the number of bits that Alice and Bob need to exchange in order to compute h.

▶ **Definition 4** (Communication complexity). Let $\mathcal{A}_\delta$ be the set of two-party algorithms that use public randomness (denoted by pub), which when used by Alice and Bob, compute h on any input a (to Alice) and b (to Bob) with an error probability smaller than δ. Let $A \in \mathcal{A}_\delta$ be an algorithm that computes h. Denote by $R_\delta^{cc-pub}(A(a,b))$ the communication complexity (denoted by cc) representing the number of 1-bit messages exchanged by Alice and Bob while executing algorithm A on a and b. We define

$$R_\delta^{cc-pub}(h) = \min_{A \in \mathcal{A}_\delta} \max_{a,b \in \{0,1\}^k} R_\delta^{cc-pub}(A(a,b))$$

to be the smallest amount of bits any algorithm would need to send in order to compute h.

A well-studied problem in communication complexity is that of set disjointness, where we are given two subsets of $\{0,\dots,k-1\}$ and need to decide whether they are disjoint. Here, the strings a and b indicate membership of elements to each of these sets.

▶ **Definition 5** (Disjointness problem). The set disjointness function $\mathrm{disj_k} : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}$ is defined as follows.

$$\mathrm{disj_k}(a,b) = \begin{cases} 0 & : \text{if there is an } i \in \{0,\dots,k-1\} \text{ such that } a(i) = b(i) = 1 \\ 1 & : \text{otherwise} \end{cases}$$

where $a(i)$ and $b(i)$ are the i-th bit of a and b respectively (indicating whether an element is a member of the corresponding set.)

We use the following basic theorem that was proven in Example 3.22 in [20] and in [2, 3, 18, 31].

▶ **Theorem 6.** *For any sufficiently small $\delta > 0$ we can bound $R_{\delta}^{cc-pub}(\mathrm{disj_k})$ by $\Omega(k)$.*

4.2 Lower Bounds for Weighted Diameter Computation in the Unicast CONGEST Model

▶ **Theorem 7.** *For any $n \geq 10$ and $B \geq 1$ and sufficiently small ε any distributed randomized ε-error algorithm A that computes a $(2 - 1/poly(n))$-approximation of the diameter of a positively weighted graph requires at least $\Omega(\frac{n}{B})$ time for some n-node graph.*

We follow the strategy of [10] and reduce the function $disj_{k(n)^2}$ to finding the diameter of a graph G. Note that the graph in [10] is unweighted, while ours is weighted. We set a parameter $k(n)$ to be $k(n) = \lfloor \frac{n}{10} \rfloor$ and construct a graph $G_{a,b}$. We do so by defining a graph $G_a = (V_a, E_a)$ that depends on inputs a and a graph $G_b = (V_b, E_b)$ that depends on b. Based on these graphs G_a and G_b, we derive the graph $G_{a,b}$ containing both G_a and G_b. We start by constructing sets of nodes $L = \{l_v | v \in \{1, \ldots, 2k(n) - 1\}\}$ and $R = \{r_v | v \in \{1, \ldots, 2k(n) - 1\}\}$. Let $L_1 = \{l_v | v \in \{1, \ldots, k(n) - 1\}\}$ and $L_2 = \{l_v | v \in \{k(n), \ldots, 2k(n) - 1\}\}$, and define $R_1 = \{r_v | v \in \{1, \ldots, k(n) - 1\}\}$ and $R_2 = \{r_v | v \in \{k(n), \ldots, 2k(n) - 1\}\}$. We add a node c_L to V_a and a node c_R to V_b, then add edges from c_L to all nodes in L and from c_R to all nodes in R. We also add edges between each pair of nodes in L_1, R_1, L_2, and R_2, and from l_i to r_i for $i \in \{1, \ldots, 2k(n) - 1\}$. Finally, we add an edge from c_L to c_R. Note that these sets of (right/left) nodes only depend on the lengths of the inputs. In the proof we define edges E_a that connect nodes in V_a depending on a. We also define edges E_b that connect nodes in V_b depending on b.

As in [10], we can represent the $k(n)^2 - 1$ bits of input a by the $k(n)^2$ possible edges between the $k(n)$ nodes L_1 and $k(n)$ nodes L_2. More specifically, we choose the mapping from integers in $\{1, \ldots, k(n)^2 - 1\}$ to pairs of integers in $\{1, \ldots, k(n) - 1\} \times \{k(n), \ldots, 2k(n) - 1\}$, such that i is mapped to $(l_{u_i}, l_{v_i}) = \left(i \mod k(n), k(n) + \left\lfloor \frac{i}{k(n)} \right\rfloor\right)$. We add edge (l_{u_i}, l_{v_i}) to G_a if and only if $a(i) = 0$, and likewise represent the bits of b by adding edge (r_{u_i}, r_{v_i}) to G_b if and only if $b(i) = 0$.

We call the graph defined by these edges $G_a = (V_a, E_a)$, and construct a similar graph G_b for input b. We define the cut-set $C_{k(n)^2} = \{(l_v, r_v) : v \in \{0, \ldots, 2k(n) - 1\}\}$ to be the $2k(n)$ edges connecting each l_v to the corresponding r_v. We will refer to the sets of vertices $L_1 \cup R_1 = \{l_v | v \in \{1, \ldots, k(n) - 1\}\} \cup \{r_v | v \in \{1, \ldots, k(n) - 1\}\}$ as **UP** (upper part of the graph) and $L_2 \cup R_2 = \{l_v | v \in \{k(n), \ldots, 2k(n) - 1\}\} \cup \{r_v | v \in \{k(n), \ldots, 2k(n) - 1\}\}$ as **LP** (lower part of the graph).

Figure 1 visualize this and we note that the former set is in the upper portion of the graph, and the latter is in the lower portion. Finally, we set $G_{a,b} = G_a \cup G_b \cup C_k$.

Now we assign weights to the edges in this construction. We set the weight of every edge in G_a and in G_b to be 1, and the weight of each edge in $C_{k(n)^2}$ to be $1/p$, the smallest possible weight (see definition of the weights in Section 3).

▶ **Lemma 8.** *The weighted diameter of $G_{a,b}$ is at most $2 + 1/p$.*

Proof. We show case by case that for any nodes u and v in $G_{a,b}$ the distance $d_w(u, v)$ is at most $2 + 1/p$. The cases are as follows:

1. **Nodes u and v are both in G_a:** Every node in G_a other than C_L is connected to C_L by an edge of length 1, and thus each node in G_a can reach any other node in G_a using at most two edges of length 1. Thus, $d_w(u, v) \leq d_w(u, c_L) + d_w(c_L, v) \leq 2$.

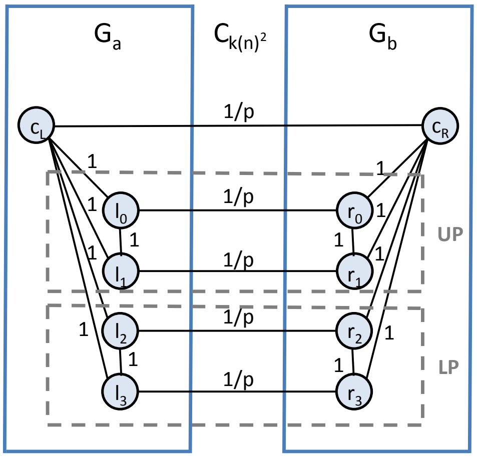


Figure 1 Base graph of (weighted) diameter $2 + 1/p$.

2. **Nodes u and v are both in G_b:** This case is identical to the previous case, so $d_w(u, v) \leq 2$.
3. **Node u is in G_a and node v is in G_b (or vice verse):** From u it is at most one hop to C_L of length 1, and from v it is at most one hop to C_R of length 1. Since the edge between c_L and c_R has weight $1/p$, we conclude that $d_w(u, v) \leq d_w(u, c_L) + d_w(c_L, c_R) + d_w(c_R, v) = 2 + 1/p$.

◀

Following the ideas of [10], we reduce the problem of deciding disjointness between sets a and b to computing the diameter of a graph.

▶ **Lemma 9.** *The diameter of $G_{a,b}$ is $1 + 2/p$ if the sets a and b are disjoint, else it is $2 + 1/p$.*

Proof. **If inputs a and b are not disjoint,** then there exists an $i \in \{1, \ldots, k(n)^2\}$ such that $a(i) = b(i) = 1$. Let us fix such an i for now and let $\nu := i \mod k(n)$ and $\mu := k(n) + \left\lfloor \frac{i}{k(n)} \right\rfloor$. We show that the two nodes l_ν and r_μ have distance of at least $2 + 1/p$. The path must contain an edge of length $1/p$ from the cut-set $C_{k(n)^2}$, since these are the only edges that connect G_a to G_b. To obtain a path of length $1 + 1/p$ we are only allowed to add one more edge from either G_a or G_b. When looking at the construction, the only two paths of length $1 + 1/p$ that we could hope for are (l_ν, l_μ, r_μ) and (l_ν, r_ν, r_μ). However, due to $a(i) = b(i) = 1$ and the implied choice of ν and μ, we know that the construction of $G_{a,b}$ does not include edge (l_ν, l_μ) nor edge (r_ν, r_μ). Thus none of these paths exists and we conclude that $d_w(l_\nu, r_\mu) \geq 2 + 1/p$. **Conversely if a and b are disjoint,** the diameter of $G_{a,b}$ is at most $1 + 2/p$. We prove this by showing that for any nodes u and v in $G_{a,b}$ the distance $d_w(u, v)$ is at most $1 + 2/p$. To do this we distinguish three cases:

1. **Node u is in G_a and node v is in G_b (or vice versa):** When considering the nodes c_L and c_R, we notice that from each of these nodes every other node in the graph can be reached within 2 hops, one of which has weight $1/p$. Now we can assume without loss of generality that $u = l_\nu \in G_a$ and $v = r_\mu \in G_b$ for some $\mu, \nu \in \{1, \ldots, 2k(n) - 1\}$. Since we assumed that a and b are disjoint there must be either at least one of the edges (l_ν, l_μ) or (r_ν, r_μ) in case that one of the nodes is in **UP** and the other node is in **LP**. Thus there is at least one of the paths (l_ν, l_μ, r_μ) or (l_ν, r_ν, r_μ) with $d_w(l_\nu, r_\mu) \leq 1 + 1/p$. In the remaining case u, v are both in **UP** or both in **LP**, and we make use of the clique-edges (among

nodes in the G_a (or G_b) part of **UP** (or **LP**), there are 4 cliques in total) and conclude that u and v are connected by path (l_ν, r_ν, r_μ) of length $d_w(l_\nu, r_\nu) + d_w(r_\nu, r_\mu) = 1 + 1/p$.

2. **Nodes u and v are both in G_a:** Let $u = a_i$ and $v = a_j$. If an edge between u and v does not directly exist, then we know an edge between b_i and b_j must exist. Thus we can get from u to v by using the edges (u, b_i), (b_i, b_j) and (b_j, v), for a path of total length $1 + 2/p$.
3. **Nodes u and v are both in G_b:** we use identical logic to case 2, where both u and v are in G_a; the distance between these nodes is at most $1 + 2/p$.

Finally note, that these two cases combined with the upper bound from Lemma 8 imply that $d_w(l_\nu, r_\mu) = 2 + 1/p$ if and only if a and b are not disjoint. ◀

▶ **Lemma 10.** *Computing a $(2 - o(1))$-approximation of the diameter in positively weighted graphs requires the exchange of $\Omega(n^2)$ bits of information.*

Proof. This follows immediately from Theorem 6; computing $disj_{k(n)^2}$ requires the exchange of $\Omega(k(n)^2) = \Omega(n^2)$ bits of information through the edges in $C_{k(n)}$ in order to decide if a and b are disjoint. ◀

Proof of Theorem 7. We use the graph $G_{a,b}$ constructed above to show that any algorithm A that computes a $(2 - 1/p')$-approximation of the diameter requires $\Theta(\frac{n}{B})$ time, for a p' that we define later in terms of p.

First note that in case the diameter is $(1 + 2/p)$ any A must output a value of at most $(1 + 2/p)(2 - 3/p) = 2 + 1/p - 6/p^2$. As this value is strictly smaller than the other possible diameter of $G_{a,b}$, which is $(2 + 1/p)$, any $(2 - 3/p)$-approximation algorithm can decide whether the $D_w(G_{a,b})$ is $(1 + 1/p)$ or $(2 + 1/p)$. We set $p = 3 \cdot p'$ to get our desired $(2 - 1/p')$-approximation algorithm. Based on this one can decide if inputs a and b, that were used to construct the graph $G_{a,b}$, are disjoint.

We know due to Theorem 6 that any algorithm must exchange $\Omega(k(n)^2)$ bits of information through the edges in $C_{k(n)}$ in order to decide if a and b are disjoint. As the bandwidth of $C_{k(n)}$ is $\mathcal{O}(|C_{k(n)}| \cdot B) = \mathcal{O}(k(n) \cdot B)$, we conclude that $\Omega(k(n)/B)$ rounds are necessary to do so. Due to the choice of $k(n)$ we conclude that $\Omega(\frac{n}{B})$ rounds are necessary to $(2 - 1/p)$-approximate the diameter of a graph. ◀

4.3 Lower Bounds for Weighted Diameter Computation in the BCC Model

Given a two-party communication problem f' with inputs a, b, we define the *f'-derived graph* $G_{a,b}$ to be a graph constructed from f' as described previously, with G_a encoding the input a to one party and G_b encoding the input b to two parties.

▶ **Theorem 11.** *Given a two-party communication problem f' with inputs a, b, if $R_\epsilon^{cc-pub}(f')$ is a lower bound on the number of bits that must be communicated in f', then solving the problem on the f'-derived graph $G_{a,b}$ with a randomized algorithm A must take at least $\frac{R_\epsilon^{cc-pub}(f')}{nB}$ rounds in the* BCC *model.*

Before presenting the proof, we want to stress that the output our algorithm A depends only on the edges of $G_{a,b}$. Other edges of the clique not mentioned in the construction of $G_{a,b}$ are still present in the CONGEST-CLIQUE model (not in the CONGEST model studied in Theorem 7) but can be used only for communication. These (additional) communication edges are not part of the lower bound construction and do not affect the diameter of the graph $G_{a,b}$.

Proof. In each round, any algorithm can send at most $|G_a| \cdot B$ bits of information from G_a to G_b, as each vertex in G_a must broadcast the same B bits to all other vertices in G_b in the BCC model. Similarly, any algorithm can send at most $|G_b| \cdot B$ bits from G_b to G_a. There are no further nodes outside of $G_{a,b}$ that could increase the bandwidth. Thus, any algorithm can exchange at most $(|G_a| + |G_b|) \cdot B = nB$ bits between G_a and G_b in each round. Therefore $\frac{R_\epsilon^{cc-pub}(f')}{nB}$ is a lower bound on the number of rounds that algorithm A must take. ◀

▶ **Theorem 12.** *Computing a $(2 - o(1))$-approximation of the diameter in positively weighted graphs in the* BCC *model takes $\Omega(n/B)$ rounds.*

Proof. Computing a $(2 - o(1))$-approximation of the diameter in positively weighted graphs is shown to require the exchange of $\Omega(k(n)^2)$ bits of information at the end of the proof of Theorem 10 above. Due to the choice of $k(n)$, these are $\Omega(n^2)$ bits. The statement then follows directly from an application of Theorem 11. ◀

▶ **Theorem 13.** *Computing the diameter exactly in unweighted graphs takes $\Omega(n/B)$ in the* BCC *model.*

Proof. Computing the exact diameter of unweighted version of the graph $G_{a,b}$ is shown to require $\Omega(n^2/B)$ bits of information to be exchanged in [10]. Thus, the result follows by Theorem 11 using similar arguments as in the proof of Theorem 12. ◀

▶ Remark 14. Note that a 2-approximation of the diameter of positively weighted graphs is achievable by computing SSSP starting in an arbitrary node, and returning twice the length of the largest distance computed. To compute (exact) SSSP we can use the SSSP-algorithm presented in [25], that runs in $\tilde{O}(\sqrt{n})$ time.

5 Deterministic Hitting Set Computation in the BCC Model

▶ **Definition 15.** Given a node $u \in V$, the set $S^k(u)$ of a node $u \in G$ contains the k nodes closest to u in a weighted graph G, with ties broken by node ID. In other words, $S^k(u) \subset V$ has the following properties:

1. $|S^k(u)| = k$, and
2. for all $s \in S^k(u)$ and $t \notin S^k(u)$, either (i) $d_w(u, s) < d_w(u, t)$, or (ii) $d_w(u, s) = d_w(u, t)$ and the ID of s is smaller than the ID of t.

▶ **Definition 16.** A k-hitting set S of a graph $G = (V, E)$ is a set of nodes such that, for every node $v \in V$, there is at least one node of S in $S^k(v)$.

Our algorithm (see Appendix A.3 of the full version [16] for pseudocode) takes as input a graph G and an integer k, and returns a k-hitting set $S \subseteq V$. The algorithm works as follows: each node starts by broadcasting its k incident edges of smallest weight to all other nodes. If the node does have less than k neighbors, it just broadcasts the weight of all its incident edges. This enables every node u to locally compute a set $S^k(u)$, consisting of the k closest nodes to u in G ([25], Observation 3.12). By closest we refer to the weighted distance of nodes to u; note that $S^k(u)$ always has k nodes for any $k \leq n$, as the graph is connected. We initialize $S := \emptyset$; S is updated over time until it is our desired k-hitting set. Let at any time R be composed of the sets $S^k(v)$ such that $S^k(v) \cap S = \emptyset$ (initially R contains all $S^k(v)$). We repeatedly find the vertex v_{max} that is contained in the largest number of elements in R (breaking ties by minimum node ID). We then add this v_{max} to S and update

R accordingly. In Lemma 17 we show that this method of greedily constructing a hitting set achieves a $\mathcal{O}(\log n)$-approximation of the smallest possible hitting set.

▶ **Lemma 17.** *Given a graph G, if the smallest possible hitting set uses N vertices, then S contains at most $\mathcal{O}(N \log n)$ vertices.*

Proof. This proof can be found in Appendix A.3 of the full version of this paper [16]. ◀

▶ **Lemma 18.** *Procedure* HittingSet *described in Appendix A.3 of the full version of this paper* [16] *computes a k-hitting set of size $\tilde{\mathcal{O}}(n/k)$ in $\mathcal{O}(k)$ rounds.*[4]

Proof. See Appendix A.3 of [16]. ◀

6 Deterministic $(2 + o(1))$-Approximation of APSP in Time $\tilde{\mathcal{O}}(n^{1/2})$ in the BCC Model

Nanongkai provides a randomized distributed algorithm ([25], Algorithm 5.2) to $(2 + o(1))$-approximate APSP in the BCC model that runs in $\tilde{\mathcal{O}}(n^{1/2})$ time. At a high level, this algorithm works by

1. choosing a random $\sqrt{n}$*-hitting set* $R \subseteq V$ of size $\tilde{\mathcal{O}}(\sqrt{n})$ such that for all nodes in V, there is some node in R within $\sqrt{n}$ hops,
2. $(1 + o(1))$-approximate (using random delays to avoid congestion) shortest paths from each node in the hitting set R to every node in V,
3. using these shortest paths to approximate shortest paths between all pairs of nodes.

We have previously presented a method to deterministically compute a $\sqrt{n}$-hitting set $R \subseteq V$ in Section 5. Nanongkai uses a randomized procedure to compute shortest paths from this hitting set to all other nodes, which we will replace by a deterministic one in this paper. This results in a deterministic $\tilde{\mathcal{O}}(n^{1/2})$ round algorithm:

▶ **Theorem 19.** *The deterministic Algorithm 2 (stated fully in* [16]*) returns a $(2 + o(1))$-approximation of APSP in time $\tilde{\mathcal{O}}(n^{1/2})$.*

The remainder of this section is devoted to explaining and analyzing this algorithm in order to prove this theorem (see pseudocode in Appendix A.2 of the full version [16]). While doing so, we also review the majority of Algorithm 5.2 of [26]. We do this to be able to point out our modifications exactly and to argue that each step can indeed be done in the BCC model, while the original implementation of Algorithm 5.2 of [26] is just stated for the CONGEST-CLIQUE model (without distinguishing between BCC and UCC models). As shown in Theorem 5.3 of [26], Algorithm 5.2 of [26] computes a $(2 + o(1))$-approximation of APSP on weighted graphs. Note that we only change the implementation of the algorithm to be deterministic, meaning we can immediately derive the same approximation ratio (with probability 1 instead of w.h.p.).

Given a graph G, we start by computing a *k-shortcut graph* G^k of G for $k = \sqrt{n}$, defined below.

▶ **Definition 20** (k-shortcut graph)**.** The shortcut graph $G^k = (V, E^k)$ is obtained by adding an edge (u, v) of weight $d_w(u, v)$ to E^k for every $u \in V$ and $v \in S^k(u)$.

[4] By using the $\mathcal{O}$-notation we implicitly assume that $k \leq n^{1-\text{polylog}n}$, which will always be the case in this paper.

To construct this graph, each node begins by broadcasting the k lightest edges adjacent to it. If there are less than k edges adjacent to a node, that node just broadcasts all of them and their weights. Based on this information each node $u \in V$ can compute $S^k(u)$, since running e.g. k rounds of Dijkstra's algorithm will only need the k-lightest edges incident to each node (as argued in [26]). During the next $O(k)$ time steps, each node u simultaneously broadcasts its $S^k(u)$ and creates a simulated shortcut edge from every node $u \in G$ to every node $v \in S^k(u)$. New edge weights $w'(u,v) := \min\{w(u,v), \min_{z \in S^k(u)} d_w(u,z) + d_w(z,v)\}$ are assigned to this graph. Then, node u locally computes a k-hitting set R of G, as described in Section 5.

To further describe our algorithm we need the following definitions.

▶ **Definition 21** (h-hop SSSP ([25], Definition 3.1))**.** Consider a network (G,w) and a given integer h. For any nodes u and v, let $P^h(u,v)$ be the set of all (u,v)-paths containing at most h edges. Define the h-hop distance between u and v as

$$d_w^h(u,v) = \begin{cases} min_{P \in P^h(u,v)} w(P) & : \ P^h(u,v) \neq \emptyset \\ \infty & : \ otherwise. \end{cases}$$

Let h-hop SSSP be the problem where, for a given weighted network (G,w), source node s (node s knows that it is the source), and integer h (known to every node), we want every node u to know $dist^h_{G,w}(s,u)$.

▶ **Definition 22** (MSSP, h-hop MSSP [26])**.** Given a set $S \subseteq V$, the multi-source shortest paths problem (MSSP) is to compute the SSSP tree from each node in S. This problem is also referred to as the S-shortest paths problem (S-SP). In the h-hop MSSP problem (a.k.a. h-hop S-SP [7, 14]) one is interested in the h-hop versions of SSSP w.r.t source nodes S.

Nanongkai states an MSSP algorithm that works in the CONGEST model, and computes $(1 + o(1))$-approximate distances on weighted graphs. The main idea of this algorithm is based on the following theorem.

▶ **Theorem 23** ([25], Theorem 3.3)**.** *Consider any n-node weighted graph (G,w) and integer h. Let $\epsilon = 1/\log n$, and let W be the maximum-weight edge in G. For any i and edge (x,y), let $D'_i = 2^i$ and $w'_i(x,y) = \left\lceil \frac{2hw(x,y)}{\epsilon D'_i} \right\rceil$. For any nodes u and v, if we let*

$$\widetilde{d}_w^h(u,v) = min \left\{ \frac{\epsilon D'_i}{2h} \times d_{w'_i}(u,v) \mid i : d_{w'_i}(u,v) \leq (1 + 2/\epsilon)h \right\},$$

then $d_w^h(u,v) \leq \widetilde{d}_w^h(u,v) \leq (1+\epsilon) \cdot d_w^h(u,v)$.

This theorem states that we can compute an $(1+\varepsilon)$-approximation of h-hop-bounded SSSP when we run $\mathcal{O}(\log n)$ many h-hop-bounded SSSP computations rooted in node u, each with modified weights $w'_1, \ldots . w'_{\log n}$. To obtain an $(1+\varepsilon)$-approximation for h-hop-bounded MSSP for sources S, Nanongkai performs $\mathcal{O}(\log n)$ many h-hop-bounded MSSP computations rooted in S, each with modified weights $w'_1, \ldots . w'_{\log n}$. In each execution of a h-hop MSSP, Nanongkai starts all h-hop SSSP computations in all nodes of S simultaneously and delays each step of any h-hop SSSP algorithm by a random amount. This is shown to guarantee that with high probability the $|S|$ copies of h-hop SSSP do not conflict with each other.

We can adapt Nanongkai's h-hop MSSP algorithm to a deterministic setting using the source detection algorithm of [21].

▶ **Definition 24** ((S, H, K)-source detection [21]). Given an unweighted graph G and $H, K \in \mathbb{N}_0$, the (S, H, K)-source detection problem is to output for each node $u \in V$ the set $L_u(H, K)$ of all (up to) K closest sources in S to u, which are at most H hops away.

▶ **Lemma 25** (Theorem 4.4, [21]). *The (S, H, K)-source detection problem can be solved in the* CONGEST *model in* $\min(H, D) + \min(K, |S|)$ *rounds.*

In Algorithm 1 of [21] (which corresponds to Lemma 25), each node always broadcasts the same message within each time step to all neighbors. Therefore it runs in the broadcast version of the CONGEST model. The algorithm is stated for unweighted graphs; we adapt it to weighted graphs by replacing every edge e of weight $w(e)$ by a path of $w(e)$ edges, each of weight 1. The simulation of these new nodes and edges is handled by the two nodes adjacent to e, and is equivalent to delaying any transmission through e by $w(e)$ rounds as it is done in [25]. This transforms a weighted graph into an unweighted one.

We now use the above deterministic procedure instead of Nanongkai's randomized one to approximate weighted h-hop MSSP on the hitting set by choosing $S := R$. In each execution of the unweighted h-hop MSSP on R, during iteration i, we set the weight $w'_i(x, y)$ to be $\left\lceil \frac{2hw'(x,y)}{\epsilon 2^i} \right\rceil$, then we execute Lenzen and Peleg's (S, H, K)-source detection algorithm (Lemma 25) on graph G^k using weight w'_i with $R := S$ and $H := h$. Furthermore, we set $K := |R|$ to guarantee that all sources within h hops are detected. We use the fact that, in our model, nodes at any distance in the graph G can directly communicate with each other, so $|D| = 1$ and $min(H, D) = h$.

After all $\mathcal{O}(\log n)$ executions of the source-detection algorithm have completed, each node $u \in V$ knows its distance to every node in R under every set of weights w_i. By Theorem 23, this allows us to compute a $(1 + o(1))$-approximation of $d^h_w(s, u)$ on G^k when choosing $\varepsilon = 1/\log n$, which according to [26] is equal to $d_w(s, u)$ for each $s \in R$ and $u \in V$. This is proven in [26] via the choice of h and k, which we do not change. Finally we broadcast these weights and compute like in [26] the value $d''(u, v)$, which is shown in [26] to be a $(2 + o(1))$-approximation.

Proof of Theorem 19. *Runtime*: Broadcasting the k lowest-weight edges, one by one in each round, takes k rounds in the BCC model. Computing $S^k(u)$ and w' takes no additional communication. By Lemma 17 we can compute the k-hitting set R is computed in time $\mathcal{O}(k)$ in the BCC model. Computing weights w'_i in takes $\mathcal{O}(\log W)$ rounds. Each execution of $(R, h, |R|)$-source detection takes $h + |R|$ time steps on the (simulated) undirected graph, and there are $\mathcal{O}(\log W)$ iterations, see Lemma 25. Since $h = \mathcal{O}(n^{1/2})$ and $|R| = \tilde{\mathcal{O}}(n/k) = \tilde{\mathcal{O}}(\sqrt{n})$ (see Lemma 17) and $\log W = \mathcal{O}(\log n)$, as $W \in \text{poly } n$. The remaining parts of the algorithm only perform broadcasts, which take $|R| = \tilde{\mathcal{O}}(\sqrt{n})$ rounds. Therefore the total runtime is $\tilde{\mathcal{O}}(\sqrt{n})$.

Approximation ratio: The $(2 + o(1))$-approximation ratio for our algorithm is immediately derived from [25], as we do not change Nanongkai's algorithm besides modifying it to execute deterministically. ◀

7 Open Problems

It is natural to ask whether our method of proving lower bounds for the diameter in the BCC model can be extended to other problems. Of particular interest are those discussed in [10], since these problems use similar graph constructions for proving lower bounds. It would also be of interest to further reduce the runtime of approximating APSP in the BCC and UCC model, maybe also at the cost of larger approximation factors.

References

1 N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. J. Comput. and Syst. Sciences, 58(1):137-147, 1999. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

2 László Babai, Peter Frankl, and Janos Simon. Complexity classes in communication complexity theory (preliminary version). In *Proceedings of the 27th annual IEEE Symposium on Foundations of Computer Science, FOCS 1986, Toronto, Ontario, Canada, 27-29 October 1986*, pages 337–347, 1986.

3 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Science*, 68(4):702–732, 2004. doi:10.1016/j.jcss.2003.11.006.

4 Keren Censor-Hillel, Petteri Kaski, Janne Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. *arXiv preprint 1503.04963*, 2015.

5 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.

6 Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM (CACM)*, 51(1):107–113, 2008.

7 Benjamin Dissler. Efficient multi-aggregation with applications to centrality computation. Semester thesis, ETH Zürich, Department of Information Technology and Electrical Engineering, Zürich, Switzerland, 2013.

8 Shahar Dobzinski, Noam Nisan, and Sigal Oren. Economic efficiency requires interaction. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 – June 03, 2014*, pages 233–242, 2014.

9 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proc. of the 33rd annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2014, Paris, France, July 15-18, 2014*, pages 367–376, 2014.

10 S. Frischknecht, S. Holzer, and R. Wattenhofer. Networks cannot compute their diameter in sublinear time. In Yuval Rabani, editor, *Proc. of the 23rd Annual ACM-SIAM Symp. on Discrete Algorithms, SODA 2012*, pages 1150–1162, 2012.

11 O. Goldreich and A. Warning. Secure multi-party computation, 1998. Unpublished manuscript.

12 James W. Hegeman and Sriram V. Pemmaraju. Lessons from the congested clique applied to mapreduce. In Magnús M. Halldórsson, editor, *Structural Information and Communication Complexity – 21st International Colloquium, SIROCCO 2014, Takayama, Japan, July 23-25, 2014. Proceedings*, volume 8576 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2014.

13 James W. Hegeman, Sriram V. Pemmaraju, and Vivek Sardeshmukh. Near-constant-time distributed algorithms on a congested clique. In *DISC*, pages 514–530, 2014. doi:10.1007/978-3-662-45174-8_35.

14 S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In Darek Kowalski and Alessandro Panconesi, editors, *Proceedings of the 31st annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2012, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 355–364, 2012.

15 Stephan Holzer. *Distance Computation, Information Dissemination, and Wireless Capacity in Networks, Diss. ETH No. 21444.* Phd thesis, ETH Zurich, Zurich, Switzerland, 2013.

16 Stephan Holzer and Nathan Pinsker. Approximation of distances and shortest paths in the broadcast congest clique. *CoRR*, abs/1412.3445, 2014. URL: `http://arxiv.org/abs/1412.3445`.

17 Taisuke Izumi and Roger Wattenhofer. Time lower bounds for distributed distance oracles. In *Principles of Distributed Systems*, pages 60–75. Springer, 2014.

18 Bala Kalyanasundaram and Georg Schnitger. The Probabilistic Communication Complexity of Set Intersection. *SIAM Journal of Discrete Mathematics*, 5(4):545–557, 1992. `doi:10.1137/0405044`.

19 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. The distributed complexity of large-scale graph processing. *arXiv preprint arXiv:1311.6209 (to appear at SODA'15)*, 2013.

20 E. Kushilevitz and N. Nisan. *Communication complexity.* Cambridge University Press, Cambridge, UK, 1997.

21 C. Lenzen and D. Peleg. Efficient distributed source detection with limited bandwidth. In Panagiota Fatourou and Gadi Taubenfeld, editors, *Proceedings of the 32nd annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2013, Montreal, Quebec, Canada, July 22-24, 2013*, pages 375–382, 2013.

22 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In Panagiota Fatourou and Gadi Taubenfeld, editors, *Proceedings of the 32nd annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2013, Montreal, Quebec, Canada, July 22-24, 2013*, pages 42–50, 2013.

23 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in O(log log n) communication rounds. In *Proceedings of the 15th annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2003, San Diego, California, USA, June 7-9,2003*, pages 94–100, 2003.

24 Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

25 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC'14, pages 565–573, 2014.

26 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. *CoRR*, abs/1403.5171, 2014. URL: `http://arxiv.org/abs/1403.5171`.

27 Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen S Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *OSDI*, pages 1–15, 2012.

28 Boaz Patt-Shamir and Marat Teplitsky. The round complexity of distributed sorting: Extended abstract. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC'11, pages 249–256, New York, NY, USA, 2011. ACM.

29 David Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, USA, 2000.

30 Sriram V. Pemmaraju and Vivek B. Sardeshmukh. Algebrisation in distributed graph algorithms: Fast matrix multiplication in the congested clique. *arXiv preprint 1412.2109*, 2014.

31 Alexander A. Razborov. On the Distributional Complexity of Disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

32 Roger Wattenhofer. Principles of Distributed Computing, Lecture 11, ETH Zurich, Zurich, Switzerland, http://dcg.ethz.ch/lectures/podc_allstars/lecture/chapter11.pdf, 2011.

The Cost of Global Broadcast in Dynamic Radio Networks[*]

Mohamad Ahmadi[1], Abdolhamid Ghodselahi[2], Fabian Kuhn[3], and Anisur Rahaman Molla[4]

1 **Department of Computer Science, University of Freiburg, Freiburg, Germany**
`mahmadi@cs.uni-freiburg.de`

2 **Department of Computer Science, University of Freiburg, Freiburg, Germany**
`hghods@cs.uni-freiburg.de`

3 **Department of Computer Science, University of Freiburg, Freiburg, Germany**
`kuhn@cs.uni-freiburg.de`

4 **Department of Computer Science, University of Freiburg, Freiburg, Germany**
`armolla@cs.uni-freiburg.de`

Abstract

We study the single-message broadcast problem in dynamic radio networks. We show that the time complexity of the problem depends on the amount of stability and connectivity of the dynamic network topology and on the adaptiveness of the adversary providing the dynamic topology. More formally, we model communication using the standard graph-based radio network model. To model the dynamic network, we use a variant of the synchronous dynamic graph model introduced in [Kuhn et al., STOC 2010]. For integer parameters $T \geq 1$ and $k \geq 1$, we call a dynamic graph T-interval k-connected if for every interval of T consecutive rounds, there exists a k-vertex-connected stable subgraph. Further, for an integer parameter $\tau \geq 0$, we say that the adversary providing the dynamic network is τ-oblivious if for constructing the graph of some round t, the adversary has access to all the randomness (and states) of the algorithm up to round $t - \tau$.

As our main result, we show that for any $T \geq 1$, any $k \geq 1$, and any $\tau \geq 1$, for a τ-oblivious adversary, there is a distributed algorithm to broadcast a single message in time $O\big(\big(1 + \frac{n}{k \cdot \min\{\tau, T\}}\big) \cdot n \log^3 n\big)$. We further show that even for large interval k-connectivity, efficient broadcast is not possible for the usual adaptive adversaries. For a 1-oblivious adversary, we show that even for any $T \leq (n/k)^{1-\varepsilon}$ (for any constant $\varepsilon > 0$) and for any $k \geq 1$, global broadcast in T-interval k-connected networks requires at least $\Omega(n^2/k^2 \log n)$ time. Further, for a 0-oblivious adversary, broadcast cannot be solved in T-interval k-connected networks as long as $T < n - k$.

1998 ACM Subject Classification F.2.2 Analysis of Algorithms and Problem Complexity – Non-numerical Algorithms and Problems, G.2.2 Discrete Mathematics – Graph Theory

Keywords and phrases radio network, dynamic network, global broadcast, interval connectivity, hitting game

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.7

1 Introduction

By now, a rich theory on algorithms for large-scale wireless networks exists and we have a rather precise understanding of the complexity of many basic computation and communication

[*] Research supported by ERC Grant No. 336495 (ACDC).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 7; pp. 7:1–7:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

tasks for a variety of wireless network models. While many wireless communication models and modeling assumptions have been studied, to a large part, the considered models all share one basic property. Most of the existing work is based on static networks and on communication models where wireless signal reception is modeled in a completely deterministic way. For example, in the classic radio network model, a wireless network is modeled as a graph and a node in the graph can receive a message transmitted by some neighbor if and only if no other neighbor transmits at the same time, e.g., [5, 21]. In the SINR (or physical) model, nodes have fixed coordinates in some geometric space and a transmitted signal can be successfully received if and only if the signal-to-noise-and-interference ratio at the receiver is above a certain fixed threshold, e.g., [14, 22].

The situation in actual networks however is quite different and wireless signal reception might behave in a rather unpredictable way. There can be multiple sources for interference which cannot be controlled by a distributed algorithm and signal propagation depends on various properties of the environment. As a result, we often obtain wireless communication links with unreliable behavior [16, 24, 26, 27, 28]. In addition, wireless devices might be mobile leading to a potentially completely dynamic network topology.

As a consequence, in recent years, researchers in the wireless algorithms community have also started to consider radio network models which exhibit nondeterministic behavior and sometimes general dynamic topologies, e.g., [9, 11, 12, 2, 17, 13]. In the present paper, we continue this line of research and study the global broadcast problem in dynamic radio networks for a range of modeling assumptions. Note that in ordinary, static radio networks, albeit appearingly simple, global broadcast is one of the best studied problems in the area, (see, e.g., [4, 5, 10, 15, 23] and many others). We model a dynamic network by applying the dynamic network model introduced in [18]. Time is divided into synchronous rounds and a wireless network is modeled as a dynamic graph with a fixed set of n nodes and a set of edges which can change from round to round. For two parameters $T \geq 1$ and $k \geq 1$, a dynamic graph is called *T-interval k-connected* if for any interval of T consecutive rounds, the set of edges which are present throughout these T rounds induces a graph with vertex connectivity at least k (in [18], the model was only introduced for $k = 1$). We refer to [19] for a more thorough discussion of the model of [18] and of several earlier related dynamic network models (e.g., [3, 6, 11, 25]).

Communication is modeled by using the standard radio network model. In each round, each node can either transmit a message or listen. A listening node successfully receives a message transmitted by a neighbor in the current graph if and only if no other neighbor transmits in the same round. We assume that nodes cannot detect collisions, i.e., whether 0 or more than 1 neighbors transmit is indistinguishable for a listening node. Note that the described dynamic network model does not only allow to model topology changes due to arbitrary node mobility. It also allows to model unreliable links where the presence/availability can change for various reasons.

We assume that the dynamic graph is provided by a worst-case adversary. As we study randomized distributed protocols, we need to specify to what extent the adversary can adapt to the random decisions of the nodes when determining the sequence of network topologies. For the adaptiveness of the adversary, we use a more fine-grained classification than what is usually done. For an integer parameter $\tau \geq 0$, we say that the adversary is τ-oblivious if for determining the graph in round r, the adversary knows the randomness of all nodes of all the rounds up to round $r - \tau$. Typically, only the extreme cases are studied. An adversary which does not have access to the random decisions of the algorithm (i.e., $\tau = \infty$) is called an *oblivious* adversary, whereas an adversary which has access to the randomness of the algorithm is called an *adaptive* adversary. If the adversary even has access to the randomness of the current round ($\tau = 0$), it is called *strongly adaptive*, otherwise ($\tau = 1$), it is called *weakly adaptive*. For more precise formal definitions of the modeling assumptions, we refer to Section 2.

In our paper, we consider the problem of broadcasting a single message from a source node to all the nodes of a dynamic network. The most relevant previous work in the context of the present work appeared in [11, 13, 17]. In [11], it is shown that in 1-interval 1-connected networks (i.e., the graph is connected in every round)[1], the complexity of global broadcast for a 1-oblivious adversary is $\Theta(n^2/\log n)$. In [17] and [13], ∞-interval 1-connected graphs are considered (i.e., there is a stable connected subgraph which is present throughout the whole execution). In [17], it is shown that even for a 0-oblivious adversary, it is possible to solve broadcast in $O(n\log^2 n)$ rounds and it is shown that $\Omega(n)$ rounds are necessary even if the stable connected subgraph has diameter 2. In [13], it is shown that when only assuming an ∞-oblivious adversary, the running time can be improved to $O((D + \log n)\log n)$, where D is the diameter of the stable connected subgraph. Note that in this case, the algorithm in [13] achieves essentially the same time complexity as is possible in static graphs of diameter D [5, 20, 23]. In [13], it is also shown that for a 1-oblivious adversary, $\Omega(n/\log n)$ rounds are necessary even for $D = 2$.

1.1 Contributions

In the following, we state the results of the paper. For formal details regarding problem statement and modeling not specified in the introduction, we refer to Section 2. Our main result is a randomized broadcast algorithm for the described dynamic radio network model. The algorithm (and also partly its analysis) is based on a combination of the techniques used in [11] and [17]. We prove the following main theorem.

▶ **Theorem 1.** *Let $T \geq 1$, $\tau \geq 1$, and $k \geq 1$ be positive integer parameters. Assume that the adversary is τ-oblivious. Then, in a dynamic T-interval k-connected n-node radio network, with high probability, single message broadcast can be solved in time*

$$O\left(\left(1 + \frac{n}{k \cdot \min\{\tau, T\}}\right) \cdot n\log^3 n\right).$$

▶ Remark. Note that for small and for large values of $\min\{\tau, T\}$, one can do slightly better. It is straightforward to generalize the broadcast algorithm of [11] to complete single message broadcast in time $O\left(n^2/k\log n\right)$ in 1-interval k-connected radio networks against a 1-oblivious adversary. Using the result from [17], we also know that for a sufficiently large constant c and $T \geq cn\log^2 n$, single-message broadcast can be solved in $O(n\log^2 n)$ rounds even for $\tau = 0$. Our upper bound therefore beats previous results for $\min\{\tau, T\} = \omega(\log^4 n)$ and $T = O(n\log^2 n)$.

In addition to the upper bound of Theorem 1, we also prove a lower bound which essentially shows that even for very large values of T, some relaxation on the standard adaptive adversaries is necessary in order to get an upper bound which improves with T. For $\tau = 1$, we show that at least for small k, the generalized upper bound of [11] is essentially optimal. The lower bound can be seen as a generalization of the simple $\Omega(n^2/\log n)$ lower bound for $k = 1$ and $T = 1$ proven in [11].

[1] In [11], the connectivity condition on the dynamic network is phrased differently and slightly more general.

Table 1 An overview over the existing bounds on global broadcast in the dynamic radio network model. The results marked in bold are the results of the present paper. For the $T = \infty$ results, D refers to the diameter of the stable subgraph.

interval conn.	vertex conn.	adversary	complexity
$T = 1$	$k = 1$	$\tau = 1$	$\Theta\left(n^2/\log n\right)$ [11]
$T = \infty$	$k = 1$	$\tau = 0$	$O\left(n \log^2 n\right)$ / $\Omega(n)$, D=2[17]
$T = \infty$	$k = 1$	$\tau = \infty$	$O((D + \log n) \cdot \log n)$[13]
$T = \infty$	$k = 1$	$\tau = 1$	$\Omega(n/\log n)$, D=2 [13]
$\boldsymbol{T \geq 1}$	$\boldsymbol{k \geq 1}$	$\boldsymbol{\tau \geq 1}$	$\boldsymbol{O\left(\left(1 + \frac{n}{k \cdot \min\{\tau, T\}}\right) \cdot n \log^3 n\right)}$
$\boldsymbol{T \leq (n/k)^{1-\varepsilon}}$	$\boldsymbol{k \geq 1}$	$\boldsymbol{\tau = 1}$	$\boldsymbol{\Omega\left(n^2/(k^2 \log n)\right)}$
$\boldsymbol{T < n - k}$	$\boldsymbol{k \geq 1}$	$\boldsymbol{\tau = 0}$	***impossible***

▶ **Theorem 2.** *For every constant $\varepsilon > 0$ and every $T \leq (n/k)^{1-\varepsilon}$, the expected time to solve single-message broadcast in T-interval k-connected radio networks against a 1-oblivious adversary is at least*

$$\Omega\left(\frac{n^2}{k^2 \log n}\right).$$

In addition, we show that unless the interval connectivity is very large, single-message broadcast cannot be solved in the presence of a strongly adaptive (0-oblivious) adversary.

▶ **Theorem 3.** *For any $k \geq 1$ and any $T < n - k$, it is not possible to solve single-message broadcast in T-interval k-connected radio networks against a 0-oblivious adversary.*

The discussion of the above result appears in Section 5. We note that the above theorem is tight in the following sense. As soon as $T \geq n - k$, global broadcast can be solved (with potentially exponential time complexity) and as soon as $T = cn \log^2 n$ for a sufficiently large constant c, we know from [17] that it can even be solved in time $O(n \log^2 n)$. All results, as well as a comparison with previous work are summarized in Table 1.

▶ Remark. In [18], interval connectivity was introduced to (in particular) study the problem of broadcasting multiple messages in a dynamic network in a standard message passing model. It is shown that interval connectivity T allows to essentially speed up multi-message broadcast by a factor of T. We find it interesting that when considering a radio network model, interval connectivity seems to provide a similar speed-up, even for broadcasting a single message. Something similar also holds for graphs with large vertex connectivity. In [7, 8], it is shown that even on static graphs, vertex connectivity k allows to speed up multi message broadcast by essentially a factor k. Here, we show that a similar speed up can be obtained in radio networks even for broadcasting a single message.

2 Model and Problem Definition

Dynamic Network. As described in Section 1, we adapt the synchronous dynamic network model of [18] to model dynamic networks.[2] Time is divided into rounds such that for all $r \geq 1$, round r starts at time $r - 1$ and ends at time r. A dynamic network is given by a

[2] Similar dynamic network models have also been used before [18], for example in [3, 11, 25]. For additional references and a thorough discussion, we refer to [19].

sequence of undirected graphs $\langle G_1, G_2, \dots \rangle$, where $G_r = (V, E_r)$ is a static graph representing the network topology in round r. The node set V is a set of n nodes corresponding to the wireless devices in the network and the edge set E_r is the set of active communication links in round r. A dynamic graph $\langle G_1, G_2, \dots \rangle$ is called *T-interval k-connected* for integer parameters $T \geq 1$ and $k \geq 1$ if and only if for all $r \geq 1$, the graph

$$\bar{G}_{r,T} = (V, \bar{E}_{r,T}), \qquad \bar{E}_{r,T} := \bigcap_{r'=r}^{r+T-1} E_{r'}$$

is a graph with vertex connectivity at least k.

Communication Model. An n-node distributed algorithm $\mathcal{A}$ is defined by n randomized processes which are assigned to the nodes of the dynamic graph by an adversary. For simplicity we use the term *node u* to also refer to the process which is assigned to node u. In each round, each node decides either to transmit a message or to listen to the wireless channel. The behavior of the wireless channel is modeled by using the standard radio network model first used in [5, 10]. When node u decides to transmit in round r, its message reaches all of its neighbours in G_r. A node v which listens in round r receives a message transmitted by a neighbor u if and only if u is the only neighbor of v in G_r which is transmitting in round r. If no message reaches v (no neighbor is transmitting), v receives silence, indicated by $\perp$. If two or more messages reach v, v also receives $\perp$, i.e., v cannot distinguish 2 or more transmitting neighbors from silence.

Adversary. We assume that the network changes under the control of an adversary. For any round r the adversary has to determine G_r based on the knowledge it has. For an integer $\tau \geq 0$, we call an adversary *τ-oblivious* if for any $r \geq 1$, the adversary constructs G_r based on the knowledge of: (1) the algorithm description, (2) the network topologies of rounds $1, \dots, r-1$, and (3) the nodes' random choices of the first $r - \tau$ rounds.

Global Broadcast. A distributed algorithm solving the global broadcast problem needs to disseminate a single message $\mathcal{M}$ from a distinguished source node to all the processes in the network. We assume that in a distributed broadcast algorithm, non-source nodes are activated (and can start to actively transmit) when they first receive the broadcast message $\mathcal{M}$. Nodes that do not yet know $\mathcal{M}$ remain silent.

Mathematical Notation. For two integers $a \leq b$, $[a, b]$ denotes the set of all integers between a and b (including a and b). Further, for an integer $a \geq 1$, we use $[a]$ as a short form to denote $[a] := [1, a]$. We say that a probability event happens with high probability (w.h.p.) if it happens with probability at least $1 - 1/n^c$, where n is the number of nodes and $c > 0$ is a constant which can be chosen arbitrarily large by adjusting other constants.

3 Upper Bound

3.1 Randomized Broadcasting Algorithm

We now describe our randomized algorithm which solves broadcast in a T-interval k-connected radio network against a τ-oblivious adversary. As stated in Section 1.1, the algorithm has a time complexity of $O\left((1 + n/(k\psi)) \cdot n \log^3 n\right)$ with high probability where $\psi := \min\{\tau, T, n/2k\}$. In light of the comment following Theorem 1 in Section 1.1, throughout

Section 3, we assume that $\psi = \Omega(\log^3 n)$ as otherwise, one can achieve a stronger upper bound by just using an adapted version of [11].

In the first round, the source node transmits the message to its neighbors. Because we assume that each graph is k-vertex connected, after one round, at least $k+1$ nodes know the message. From there on, our randomized algorithm works in *phases*. To simplify notation, in the following, we ignore the first round and assume that at time 0, the algorithm starts with at least $k+1$ nodes which know the broadcast message $\mathcal{M}$. The phases of the algorithm are defined as follows.

▶ **Definition 4** (Phase). The j^{th} time interval of ψ consecutive rounds is called phase j, where j is a positive integer. Hence, phase j starts at time $(j-1)\psi$ and ends at time $j\psi$ and it consists of rounds $(j-1)\psi+1, \ldots, j\psi$.

Let t_v denote the round in which $\mathcal{M}$ is received by node v for the first time. In each round t the set V is partitioned into following three subsets. The *previously informed nodes* $I(t)$ are the nodes that have received $\mathcal{M}$ in some phase before the current phase. Note that in the first phase, $I(t)$ consists of at least $k+1$ informed nodes. The nodes that have received $\mathcal{M}$ for the first time in the current phase in some round before time t are called *newly informed nodes*, and they are denoted by $N(t)$. Finally, the set of *uninformed nodes* at time t is denoted by $U(t) := V \setminus \{I(t) \cup N(t)\}$.

The algorithm can be seen as a combination of two existing protocols which appeared in [11] and [17]. The protocol of [11] is a very basic one where all informed nodes always try to transmit the message independently with the same uniform probability. In the *harmonic broadcast* protocol of [17], informed nodes use harmonically decaying probabilities to forward the message. In each phase of our algorithm, in the first $\lceil \psi/2 \rceil$ rounds, a variant of the protocol of [11] is applied and in the second $\lfloor \psi/2 \rfloor$ rounds, the idea of the protocol of [17] is applied. In the following, the algorithm is described in detail.

First half of a phase. In the first $\lceil \psi/2 \rceil$ rounds of a phase, all informed nodes, i.e., all $v \in \{I(t) \cup N(t)\}$, transmit the message with probability $1/n$.

Second half of a phase. The nodes in $U(t) \cup I(t)$ remain silent throughout the second half of a phase. However, in each round t, any node $v \in N(t)$ transmits the message with probability $p_v(t)$, given by

$$\forall t > \left\lfloor \frac{t}{\psi} \right\rfloor \cdot \psi + \left\lceil \frac{\psi}{2} \right\rceil \;:\; \forall v \in N(t) \;:\; p_v(t) := \frac{1}{1 + \left\lfloor \frac{t - \hat{t}_v - 1}{\mathcal{T}} \right\rfloor}, \tag{1}$$

where

$$\hat{t}_v := \begin{cases} \left\lfloor \frac{t}{\psi} \right\rfloor \cdot \psi + \left\lceil \frac{\psi}{2} \right\rceil, & \text{if } \lfloor t/\psi \rfloor \cdot \psi < t_v < \lfloor t/\psi \rfloor \cdot \psi + \lceil \psi/2 \rceil \\ t_v, & \text{otherwise} \end{cases}$$

and $\mathcal{T}$ will be fixed in Lemma 5.

Thus, in the second half of a phase, only nodes participate which for the first time receive $\mathcal{M}$ in the current phase. Each node v which gets newly informed in the phase executes the following protocol. As soon as v knows $\mathcal{M}$ and as soon as the second half of the phase has started, v starts transmitting $\mathcal{M}$ to its neighbors. For the first $\mathcal{T}$ rounds, v transmits the $\mathcal{M}$ with probability 1, for the next $\mathcal{T}$ rounds v transmits $\mathcal{M}$ with probability 1/2, and the probability for the next time intervals of $\mathcal{T}$ rounds becomes 1/3,1/4, etc.

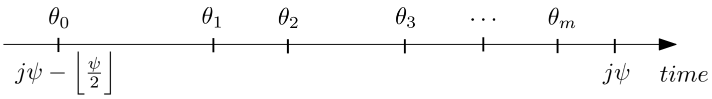


Figure 1 Time intervals $[\theta_{i-1}, \theta_i]$ with equal number of free and busy rounds where $i > 1$.

3.2 Analysis

Recall that by the definitiuon of ψ, we have $k\psi \le n/2$, $\psi \le T$, and $\psi \le \tau$. The T-interval k-connectivity of the dynamic network guarantees the existence of a stable spanning subgraph with vertex connectivity of at least k throughout the whole duration of every phase. We call this reliable spanning subgraph the *backbone* of the phase. Note that we may have different backbones in different phases. Let $P(t)$ denote the sum of transmitting probabilities of all the nodes in round t, i.e., $P(t) := \sum_{v \in V} p_v(t)$. For the analysis of our algorithm, we say that round t is *busy* if $P(t) \ge 1$ and otherwise we say that round t is *free*. If the node v is the only node transmitting in a round, we say that node v gets *isolated* in that round.

For any phase j, let $\theta_0 := j\psi - \lfloor \psi/2 \rfloor$, i.e., θ_0 is the time when the second half of the phase starts. For $i > 0$, we define $\theta_i > \theta_{i-1}$ to be the first time such that in the time interval $[\theta_{i-1}, \theta_i]$ (i.e., in rounds $\theta_{i-1}+1, \ldots, \theta_i$) the number of busy rounds equals the number of free rounds (see Figure 1).

We further define $m \ge 0$ such that θ_m is the last such time defined for a given phase. The case $m = 0$ implies that throughout the second half of the phase, the number of busy rounds is always larger than the number of free rounds.

We use the following lemma adapted from Lemma 13 of [17].

▶ **Lemma 5** ([17]). *Consider a node v. Let $t > \hat{t}_v$ be such that at least half of the rounds $\hat{t}_v + 1, \ldots, t$ are free. If $\mathcal{T} \ge \lceil 12 \ln(n/\epsilon) \rceil$ for some $\epsilon > 0$, then with probability larger than $1 - \epsilon/n$ there exists a round $t' \in \left\{\hat{t}_v + 1, \ldots, t\right\}$ such that v is isolated in round t'.*

▶ **Lemma 6.** *For all phases, in each time interval $[\theta_{i-1}, \theta_i]$, where $i \in [m]$, if round $\theta_{i-1} + 1$ is busy then any node v with $\hat{t}_v \in \{\theta_{i-1}, \ldots . \theta_i - 1\}$ gets isolated in some round $t' \in \left\{\hat{t}_v + 1, \ldots, \theta_i\right\}$ with high probability.*

Proof. Let $\bar{t}$ denote the first round that the number of free rounds equals the number of busy rounds starting from round $\hat{t}_v + 1$. For the sake of contradiction, assume that $\bar{t} > \theta_i$, that is, the number of free rounds is less than the number of busy rounds in $\left\{\hat{t}_v + 1, \ldots, \theta_i\right\}$ and we also know that the number of busy rounds is greater than the number of free rounds in $\left\{\theta_{i-1} + 1, \ldots, \hat{t}_v\right\}$ (because of minimality of θ_i and the fact that round $\theta_{i-1} + 1$ is busy). It follows that the number of busy rounds is greater than the number of free rounds in $\{\theta_{i-1} + 1, \ldots, \theta_i\}$ contradicting our assumption on the equality of free and busy rounds in $\{\theta_{i-1} + 1, \ldots, \theta_i\}$. Therefore, $\bar{t} \le \theta_i$ and according to Lemma 5 the claim holds. ◀

As one can see in Figure 2, at the beginning of each phase, the uninformed nodes in the backbone form one or several connected subgraphs which we call the *uninformed connected components*. For each uninformed connected component there must exist some edge in the backbone (within a phase) connecting an informed node to a node in that component. Note that because the adversary is τ-oblivious and thus also oblivious to the last $\psi \le \tau$ rounds, the adversary has to determine the dynamic graph throughout a phase before the phase starts. The backbone graph of a phase can therefore not change depending on the randomness of the algorithm during the phase.

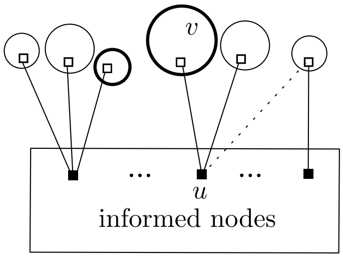


Figure 2 Backbone of a phase. Available components are identified by thick circles.

▶ **Definition 7** (Available Components and Available Nodes). At the end of the first $\lceil \psi/2 \rceil$ rounds of each phase, any uninformed connected component that includes at least one newly informed node is called an available component. All the nodes in an available component are called available nodes.

▶ **Lemma 8.** *Consider an arbitrary phase and an arbitrary $i \geq 1$. If at the beginning of round θ_i of the phase there exists at least one uninformed available node, then w.h.p. at least one available node gets informed in round θ_i.*

Proof. We will show that for every $i \geq 1$, w.h.p., if there is some node available u with $\hat{t}_u = \theta_{i-1}$ and at the beginning of round θ_i, there is at least one uninformed available node, then at least one available node v gets informed in round θ_i. The claim of the lemma then follows by induction on i. If there are no available nodes, there is nothing to prove. If there are available nodes, there is at least one node u which gets newly informed in the first half of the phase and we therefore have $\hat{t}_u = \theta_0$. Using the above claim, it then w.h.p. follows that if there still is an uninformed available node at time $\theta_1 - 1$, some uninformed available node u' gets informed in round θ_1 and thus $\hat{t}_{u'} = \theta_1$. For $i > 1$, the induction step now follows in the same way. It therefore remains to show that w.h.p., if there is some node available u with $\hat{t}_u = \theta_{i-1}$ and at the beginning of round θ_i, there is at least one uninformed available node, then at least one available node v gets informed in round θ_i.

By Lemma 6 we know that w.h.p., all the nodes u with $\hat{t}_u \in \{\theta_{i-1}, \ldots, \theta_i - 1\}$ get isolated in some round $t' \in \{\theta_{i-1} + 1, \ldots, \theta_i\}$. Hence, by induction on j, w.h.p., for all $j \leq i-1$ there is some node u' with $\hat{t}_{u'} = \theta_j$ and therefore all nodes u with $\hat{t}_u \in \{\theta_0, \ldots, \theta_i - 1\}$ get isolated in some round $t' \in \{\theta_0 + 1, \ldots, \theta_i\}$. Consequently, w.h.p., all newly informed nodes $N(\theta_i - 1)$ at time $\theta_i - 1$ get isolated in some round $t' \in \{\theta_0 + 1, \ldots, \theta_i\}$. Let v be an uninformed available node before round θ_i (i.e., at time $\theta_i - 1$). Because v is available, at time $\theta_i - 1$, there is a informed available neighbor u in the backbone graph of the current phase. We clearly have $u \in N(\theta_i - 1)$ and thus w.h.p., u gets isolated in some round $t' \in \{\theta_0 + 1, \ldots, \theta_i\}$. As soon as u gets isolated, v gets informed and we can therefore conclude that u gets isolated in round θ_i and thus v gets informed in round θ_i. ◀

The proof of the following lemma appears in the full version of this paper [1].

▶ **Lemma 9.** *Consider an arbitrary phase and assume that at the beginning of the second $\lfloor \psi/2 \rfloor$ rounds of the phase there are z available nodes. Then, w.h.p., for some constant $c > 0$, at the end of the phase we have at least $\min\left\{z, c\psi/\ln^2 n\right\}$ newly informed nodes.*

Using the established technical lemmas, we can now proof our upper bound theorem.

▶ **Theorem 1 (restated).** *Let $T \geq 1$, $\tau \geq 1$, and $k \geq 1$ be positive integer parameters. Assume that the adversary is τ-oblivious. Then, in a dynamic T-interval k-connected n-node radio network, with high probability, single message broadcast can be solved in time*

$$O\left(\left(1+\frac{n}{k\cdot\min\{\tau,T\}}\right)\cdot n\log^3 n\right).$$

Proof. Consider some phase j and let $\mathcal{B}_j$ be the backbone of phase j, i.e., $\mathcal{B}_j$ is the stable k-connected subgraph of phase j. Consider the subgraph $\mathcal{B}_j[U_j]$ of $\mathcal{B}_j$ induced by the uninformed nodes U_j. This induced subgraph might consist of several connected components. However, each of the components is connected to at least k nodes in I_j as it is shown in Figure 2 (recall that we can assume that $|I(t)| \geq k+1$). Note that if one of these at least k nodes gets isolated in the first half of the phase, all nodes in the component become available for the second half of the phase.

In the first half of phase j, in any round t, each node in $I(t)$ transmits the message with probability $1/n$. Therefore, for every node $u \in I(t)$, the probability that u gets isolated in round $t+1$ (in the first half of a phase) is at least

$$\Pr(u \text{ gets isolated in round } t+1) \geq \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1} > \frac{1}{en}. \tag{2}$$

In following, we analyze the progress in the first half of some phase j. Consider an uninformed node $v \in U_j$ (at the beginning of phase j). Let $\mathcal{A}_v$ be the event that v becomes available in the first half of phase j. Event $\mathcal{A}_v$ definitely occurs if one of the at least k initially informed neighbors of v's component in $\mathcal{B}_j[U_j]$ gets isolated in one of the at least $\psi/2$ rounds of the first half of the phase. The probability for this is

$$\Pr(\mathcal{A}_v) \geq 1-\left(1-\frac{k}{en}\right)^{\psi/2} > 1-e^{-k\psi/2en} \geq \frac{k\psi}{4en}.$$

The last inequality follows from the fact that for all $0 \leq x \leq 1$, $e^{-x} \leq 1-x/2$. Let X be the number of nodes in U_j that get available in phase j. For convenience, we define $\lambda := |U_j|/n \leq 1$. We then have

$$\mathbb{E}[X] = \sum_{v\in U_j}\Pr(\mathcal{A}_v) \geq |U_j|\cdot\frac{k\psi}{4en} = \frac{\lambda k\psi}{4e}. \tag{3}$$

We define $F := \min\left\{\frac{\psi}{16e}, \frac{c\psi}{\ln^2 n}\right\}$, where $c > 0$ is the constant that is used in Lemma 9. Note that by Lemma 9, in phase j, w.h.p., at least $\min\{X, F\}$ uninformed nodes become informed.

We define a phase to be successful if $X \geq \lambda F$. Let $\mathcal{S}$ be the event that phase j is successful and let $\bar{\mathcal{S}}$ be the complementary event. We can upper bound the expected value of X as follows:

$$\mathbb{E}[X] < \Pr(\bar{\mathcal{S}})\cdot\lambda F + \big(1-\Pr(\bar{\mathcal{S}})\big)\cdot\lambda n.$$

Combining with the upper bound in (3), we obtain (recall that we assume that $k\psi \leq n/2$).

$$\Pr(\bar{\mathcal{S}}) < \frac{n-\frac{k\psi}{4e}}{n-F} \leq \frac{n-\frac{k\psi}{4e}}{n-\frac{\psi}{16e}} \leq \left(1-\frac{k\psi}{4en}\right)\left(1+\frac{\psi}{8en}\right) \overset{(k\geq 1)}{\leq} \left(1-\frac{k\psi}{8en}\right). \tag{4}$$

By Lemma 9, in a successful phase, w.h.p., at least λF new nodes get informed. Hence, in a successful phase, w.h.p., we get rid of at least an (F/n)-fraction of the remaining uninformed nodes. In order to inform all nodes, w.h.p., we therefore need at most $O(n\log(n)/F) = O(n\log^3(n)/\psi)$ successful phases. Using (4) and a standard Chernoff argument, we can thus w.h.p. upper bound the total number of phases by $O\big(n^2\log^3(n)/(k\psi^2)\big)$. As each phase takes ψ round, this concludes the proof. ◀

4 Lower Bound

In this section we prove a lower bound for global broadcast in T-interval k-connected radio networks against a 1-oblivious adversary. Furthermore, we show impossibility of solving the same problem against a strongly adaptive adversary (0-oblivious adversary).

Our lower bound is based on a general technique for proving lower bounds for communication problems in radio networks, introduced by Newport in [23]. Using this technique, one first defines a combinatorial game for which a lower bound can be proved directly. It is then shown how to reduce the game to the problem in order to leverage the game's lower bound to obtain the desired lower bound for the problem.

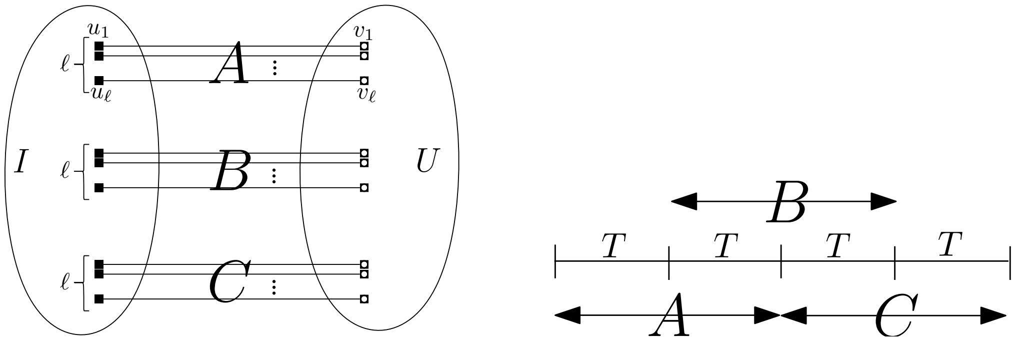
To prove Theorem 2 using this technique, we first introduce an abstract hitting game, called the (β, ℓ, φ)-*periodic hitting game* and directly prove a lower bound for winning this game. We note that this game is more involved than the games used in previous work,e.g., [13, 23]. Based on a lower bound for the hitting game, for a given instance of the game we instantiate an n-node *target network*. By instantiation of an n-node network, we mean assigning n processes with unique IDs to the nodes of the network. For the instantiation, one needs to also have information which is not available to the player in the game. However, we show that by playing the game, the player can still simulate the execution of a given broadcast algorithm on the corresponding target network to the given instance of the game. We show that this simulation of a broadcast algorithm allows to win the hitting game and the lower bound on the hitting game and the simulation together imply a lower bound for solving global broadcast.

(β, ℓ, φ)-periodic hitting game. The game is defined for three integers $\beta, \ell, \varphi > 0$ and proceeds in rounds. Time is divided into phases of φ rounds, where the j^{th} phase of the game is called phase π_j. That is, phase π_1 consists of rounds 1 to φ, phase π_2 consists of rounds $\varphi + 1$ to 2φ, etc. The first round of any phase π_j is called t_j. The player of the game is represented by a probabilistic automaton $\mathcal{P}$ and plays the game against a referee. Two sets are defined for this game, a *selection set* $S := [\beta]$ which is fixed during the game, and a *target set* which might change from round to round. The target set of round t is denoted by $X(t)$. In each round t, $\mathcal{P}$ chooses one element from S and outputs it as the guess $\gamma(t)$ for round t. Round t is called a *successful round*, if and only if $\gamma(t) \in X(t)$.

At the beginning of each phase π_j $(j \geq 1)$, the referee determines a set Y_j consisting of ℓ elements chosen uniformly at random from S. We define the target set as follows. For convenience, assume that Y_0, Y_{-1} and $X(0)$ are empty sets.

$$\forall j, \forall t \in [t_j, t_j + \varphi - 1] \, : \, X(t) := \begin{cases} Y_j \cup \left[X(t-1) \setminus (\{\gamma(t-1)\} \cup (Y_{j-2} \setminus Y_{j-1}))\right] & \text{if } t = t_j, \\ X(t-1) \setminus \{\gamma(t-1)\} & \text{if } t \neq t_j. \end{cases}$$

That is, at the beginning of each phase π_j, the referee chooses ℓ elements Y_j from S uniformly at random and adds them to the target set. Two phases (2φ rounds) later, each of these ℓ elements which still remains in the target set (and which is not in Y_{j+1}) is removed from the target set by the referee. Moreover, after each successful round, the referee removes the correct guess from the target set. Player $\mathcal{P}$ wins the game in r rounds if and only if either round r is the β^{th} successful round for $\mathcal{P}$, or before round $r + 1$ (in phase j), $X(r + 1) \cap Y_{j-1} = \emptyset$ or $X(r + 1) \cap Y_j = \emptyset$. The second condition will be used to ensure sufficiently large interval connectivity of the target network as long as the game is not won. The only information that the player receives at the end of each round is whether the round was successful or not. The player is also notified if it wins the game.



(a) The core structure of the dynamic lower bound network. The edges labeled A, B, and C are added in different phases.

(b) At least one edge exists between I and U for any T consecutive rounds.

Figure 3 A snapshot of the dynamic network used in the hitting game simulation.

Intuitively, as long as the target set changes sufficiently often, it should always appear essentially random to the player. Therefore, the best strategy for hitting the target set is to always choose an almost uniformly random guess, leading to roughly β/ℓ rounds to get a single successful round. The following lemma states this intuition formally. The proof of the lemma appears in the full version of this paper [1].

▶ **Lemma 10.** *For any $\varphi \leq \beta/3$ and for $\ell \geq \ell_0$ for a sufficiently large constant $\ell_0 > 0$, the expected number of rounds for a player to win the (β, ℓ, φ)-periodic hitting game is at least $\Omega(\beta^2/\ell)$.*

▶ **Lemma 11.** *If algorithm $\mathcal{A}$ solves the global broadcast problem in any T-interval 1-connected dynamic n-nodes network against a 1-oblivious adversary in $f(n) = n^{O(1)}$ rounds in expectation for a sufficiently large value of T, then we can construct a player $\mathcal{P}$ to win the $(\lfloor n/2 \rfloor - \ell, \ell, cT \ln n)$-periodic hitting game in expected $O(f(n) \log n)$ rounds, for some positive constants c and ℓ.*

Proof. We construct a player $\mathcal{P}$ to simulate the execution of $\mathcal{A}$ on a particular T-interval 1-connected dynamic n-node network (the target network). Then the player uses the transmitting behavior of the nodes in the simulation to generate guesses for playing the game. We start by defining the target network for a given instance of the $(\lfloor n/2 \rfloor - \ell, \ell, cT \ln n)$-periodic hitting game.

The Target Network. For the following discussion, we set $\beta := \lfloor n/2 \rfloor - \ell$ and $\varphi := cT \log n$ to denote the size of the selection set and the length of a phase of the hitting game. We assume that we are given an instance of the (β, ℓ, φ)-periodic hitting game. Based on how the hitting game develops, we define an n-node dynamic target network. We first describe the core (backbone) part of the network. The nodes of the dynamic network are defined as $V := \{0, \ldots, n-1\}$. We assume that node 0 is the source and we identify the next β nodes (i.e., the set $[\beta]$) with the selection set S of the hitting game. Throughout the execution, node 0 is connected to all nodes in $[\beta]$ and it is not connected to any other node. Throughout the simulation of the broadcast algorithm, we use I and U to denote the set of informed and uninformed nodes, respectively (a node is informed iff it knows the broadcast message $\mathcal{M}$). Clearly, as soon as the source node broadcasts $\mathcal{M}$, the set of informed nodes is $I = \{0, \ldots, \beta\}$ and we thus have $U = \{\beta + 1, \ldots, n-1\}$. To simplify notation, we assume that already at the start of the simulation, all nodes in $\{0,\dots,\beta\}$ know $\mathcal{M}$ and thus, we start round 1 with $I = \{0,\dots,\beta\}$ and $U = \{\beta+1,\dots,n-1\}$. We will assume that the number of uninformed nodes is always at least 2ℓ. As soon as it drops below, we stop carrying out the simulation.

Throughout the simulation, we always assume that all nodes in I form a clique and all nodes in U form a clique. Apart from this, the topology of the core network is determined by the target set of the hitting game that we are trying to win by simulating $\mathcal{A}$. Assume that in some round r of the hitting game, the target set is $X(r) \subset [\beta]$. During the simulation, in the backbone network we then use the nodes in $X(r)$ as bridge nodes to connect the informed nodes to the uninformed nodes. Each node $x \in X(r)$ is connected to exactly one node $n(x) \in U$ such that each node in U is connected to at most one node in $X(r)$. We assume that whenever a new node is added to $X(r)$, its neighbor in U is chosen uniformly at random among all nodes in U which are not already connected to a bridge node in $X(r)$. Note that the size of $X(r)$ is always at most 2ℓ and because we assumed that $|U| \geq 2\ell$, we can always do such an assignment of bridge nodes. Whenever the player makes a successful guess $x \in X(r)$, we move x to the set of informed nodes I and we connect x with all nodes in I and disconnect it with all nodes in the remaining set U of uninformed nodes. Note that in the hitting game, after a successful guess $x \in X(r)$, x is also removed from the target set. The target network at any time is either the described core network (backbone) or the complete graph K_n, for which the choice will be explained later.

The Simulation. The simulation of the broadcast algorithm $\mathcal{A}$ is done in a round-by-round manner. As the dynamic topology used in the simulation depends on the target set of the hitting game, the player $\mathcal{P}$ of the hitting game does not know the dynamic topology. We need to show that $\mathcal{P}$ can still correctly simulate the behavior of the broadcast algorithm.

As discussed above, we assume that at the beginning of the simulation, the set of informed nodes is $I = \{0,\dots,\beta\}$. Each round of $\mathcal{A}$ is simulated by $\mathcal{P}$ by making at most $c\ln n$ guesses in the hitting game. More specifically, a given round r of $\mathcal{A}$ is simulated as follows.

First, note that because we assume that the adversary is 1-oblivious, $\mathcal{P}$ can base the graph of round r on the states of all nodes at the beginning of the round. Hence, in particular, the graph of round r can depend on the probability $p_v(r)$ with which each node $v \in I$ transmits in the given round. We define a round r of $\mathcal{A}$ to be *busy* if $\sum_{v\in I} p_v(r) > \frac{c}{2}\ln n$, otherwise a round r is called *free*. In a busy round, the network graph is assumed to be the complete graph K_n and in a free round, the network graph is assumed to be exactly the backbone graph as described above. We assume that $\mathcal{P}$ always knows the set of informed nodes and because only informed nodes are allowed to transmit, $\mathcal{P}$ can determine all messages which are transmitted in a round by simulating the random decisions of the nodes in I. We say that the simulated execution of $\mathcal{A}$ is *bad* if either there is a free round in which more than $c\ln n$ nodes in I decide to transmit a message or if there is a busy round in which exactly one node in I decides to transmit. Otherwise, the simulated execution is called *good*. If an execution turns out to be bad, $\mathcal{P}$ stops the simulation of $\mathcal{A}$ and simply continues making random guesses until it wins the hitting game. Note that the expected time to win the hitting game in this way is at most $O(\beta^2/\ell)$ as unless there have been at least $\Omega(\ell)$ successes during the last $\phi = O(\beta)$ rounds of the hitting game, the probability for a successful guess is always $\Omega(\ell/\beta)$. As long as the number of simulated rounds $f(n)$ of $\mathcal{A}$ is polynomial in n (and thus also in β), for an arbitrary given constant $d > 0$ and sufficiently large constant c, the probability to obtain a good execution is at least $1 - 1/n^d$. For sufficiently large constant c, the expected time to win the hitting game is therefore dominated by the expected time to win the game conditioned on the event that the simulation of $\mathcal{A}$ creates a good execution.

In the following, we therefore assume that the generated broadcast execution is good. In the following, we also assume that in the current phase (of length φ) of the hitting game, there are still at least $c \ln n$ guesses that can be made. If this is not the case, player $\mathcal{P}$ first makes a sequence of unsuccessful guesses to finish the phase ($\mathcal{P}$ can for example repeat the last guess it has made before to make sure it is not successful). As we assumed that T is sufficiently large, we can assume that $\varphi \gg c \ln n$ and therefore we only waste a small fraction of all guesses by doing this.

Let us first assume that a simulated round r of $\mathcal{A}$ is busy. As in this case, the communication network is a complete graph and since we assume that in a good execution, no node gets isolated (transmits alone), every node receives silence and we therefore do not need to simulate any receive behavior. In this case, we also do not make any guesses in the hitting game. If round r is free, the number of nodes that transmit is between 0 and $c \ln n$. First recall that the nodes in I are fully connected and $\mathcal{P}$ can therefore clearly simulate their receive behavior. Further, let $Z(r) \subseteq [\beta]$ be the set of nodes in β which are transmitting in round r. For each $z \in Z(r)$, player $\mathcal{P}$ uses z as a guess in the hitting game. Note that because there are at most $c \ln n$ guesses to be made and because we assumed that there are still at least $c \ln n$ guesses in the current phase of the hitting game, during making the guesses for all $z \in Z(r)$, we do not change the phase (and thus the target set) in the hitting game. The node z therefore is a bridge node connecting I to a node $n(z) \in U$ in round r of the broadcast algorithm if and only if z is a successful guess. In that case, $n(z)$ is a uniformly random node in U. Hence, if z is a successful guess, $\mathcal{P}$ chooses $n(z)$ uniformly at random in U and it moves $n(z)$ from U to I. Note that z is also removed from the target set, and all connections of $n(z)$ to nodes in the remaining set U are removed. Note also that by choosing $n(z)$ uniformly at random in U, player $\mathcal{P}$ does not only simulate the randomness of the broadcast algorithm, but it also simulates the randomness of the adversary. As long as the execution is good, in the given dynamic network, the broadcast algorithm informs a new node if and only if one of the bridge nodes v transmits in a free round. For any bridge node v which transmits in a free round, the corresponding uninformed bridge node $n(v)$ gets informed. The described simulation therefore correctly simulates the broadcast algorithm and it informs a new node if and only if it makes a correct guess. As we only stop the simulation once the number of uninformed nodes drops below 2ℓ, we need at least $n - (\beta + 1) - 2\ell + 1 = \lceil n/2 \rceil - \ell \geq \beta$ successful guesses and thus win the game to stop the simulation.

It remains to show that the dynamic network used in the simulation is T-interval connected. Every φ guesses and thus after at least $\varphi/(c \ln n)$ rounds of the simulation, we add ℓ new edges connecting some $v \in I$ and $n(v) \in U$. As long as v is not used as a guess, such an edge remains for 2φ guesses. As long as it is always guaranteed that one of these edges survives the next 2φ guesses (and thus at least $2\varphi/(c \ln n)$ rounds), the network is at least $T = \varphi/(c \ln n)$-interval connected. Hence, the network is not guaranteed to be T-interval connected if there is a phase j in the hitting game such that all the elements of the set Y_j added to the target set at the beginning of phase j are successfully guessed by the end of phase $j + 1$. Recall that in this case, the player also wins the game and therefore the claim of the lemma follows. ◀

For $k = 1$, the statement of our main lower bound theorem (Theorem 2) now directly follows by combining Lemmas 10 and 11.

▶ **Lemma 12.** *For every constant $\varepsilon > 0$ and every $T \leq n^{1-\varepsilon}$, the expected time to solve single-message broadcast in T-interval 1-connected radio networks against a 1-oblivious adversary is at least $\Omega\big(\frac{n^2}{\log n}\big)$.*

Proof. For the sake of contradiction let us assume that $\mathcal{A}$ can solve broadcast for any T-interval 1-connected network in $f(n) = o(n^2/\ell \log n)$ rounds, then based on Lemma 11 a player can solve any instance of the $(n/2, \ell, cT \log n)$-periodic hitting game in $o(n^2/\ell)$ rounds which contradicts Lemma 10 and this proves the necessity of $\Omega(n^2/\ell \log n)$ rounds to solve broadcast in any T-interval 1-connected network. Based on Lemma 10, by choosing sufficiently large constant value for ℓ the claimed lower bound follows. ◀

In order to obtain Theorem 2, we need to generalize the above result from T-interval 1-connected networks to T-interval k-connected networks for arbitrary $k \geq 1$. As shown below, this can be achieved by using a simple generic reduction.

▶ **Theorem 2 (restated).** *For every constant $\varepsilon > 0$ and every $T \leq (n/k)^{1-\varepsilon}$, the expected time to solve single-message broadcast in T-interval k-connected radio networks against a 1-oblivious adversary is at least $\Omega\big(\frac{n^2}{k^2 \log n}\big)$.*

Proof. Given an n-node graph G, let $H_k(G)$ be the graph which is obtained by replacing each node of G by a clique of size k and by replacing each edge $\{u, v\}$ of G by a complete bipartite subgraph $K_{k,k}$ between the two k-cliques representing u and v. If G is connected, in order to disconnect $H_k(G)$ by deleting some nodes, we need to completely remove at least one of the k-cliques representing the nodes of G. Hence, if G is connected, $H_k(G)$ is k-vertex connected. It follows in the same way that if we have a dynamic graph $G_1, \dots, G_t$ which is T-interval 1-connected, the dynamic graph $H_k(G_1), \dots, H_k(G_t)$ is T-interval k-connected. Even if all nodes of such a graph $H_k(G_i)$ know to which of the cliques representing the nodes of G_i they belong, solving broadcast in the dynamic graph $H_k(G_1), \dots, H_k(G_t)$ cannot be easier than solving broadcast in $G_1, \dots, G_t$. If each graph $H_k(G_i)$ has N nodes, the graphs G_i have N/k nodes and the claimed lower bound directly follows by applying Lemma 12. ◀

5 Impossibility of Broadcast Against a 0-Oblivious Adversary

In this section, we prove the impossibility result that we stated in Section 1.1. We show that unless T is almost equal to n, the global broadcast problem cannot be solved in the presence of a 0-oblivious adversary, even for very large vertex connectivity k.

▶ **Theorem 3 (restated).** *For any $k \geq 1$ and any $T < n - k$, it is not possible to solve single-message broadcast in T-interval k-connected radio networks against a 0-oblivious adversary.*

Proof. We show that a strongly adaptive adversary (i.e., a 0-oblivious adversary) can apply a simple strategy to prevent any algorithm from solving the global broadcast problem in a T-interval k-connected network, where $T < n - k$. Consider the following adversary strategy to determine the sequence of network topologies.

The adversary partitions the n nodes into two distinct sets A and B, such that A includes the source node and is of size $T + k$, and B is of size $n - (T + k)$. Since $T < n - k$, there exists at least one node in B. Note that at the beginning, no node in B knows the broadcast message $\mathcal{M}$ (or anything about $\mathcal{M}$). Because the adversary is 0-oblivious, in each round r, it can determine the graph after all nodes have made their random decisions. It can therefore determine the graph based on which nodes transmit.

If in a round r, either 0 nodes transmit or at least 2 nodes transmit, the network graph is chosen to be the complete graph. Note that in such a round, there is either silence or all nodes experience a collision. In both cases, all listening nodes receive $\perp$ and therefore no node in B can learn something about $\mathcal{M}$.

If in a round, exactly one node v in A transmits, the network graph consists of all edges except the edges connecting v to nodes in B. Like this, also in this case all nodes in B receive $\perp$ and they therefore cannot learn something about $\mathcal{M}$.

It remains to show that the given dynamic graph is T-interval k-connected. During the whole execution, A is a clique consisting of $T+k$ nodes. Hence, A is a T-interval k-connected network. To show that whole n-node network is also a T-interval k-connected network, it is sufficient to show that for any node $v \in B$, in any T consecutive rounds, there exist at least k fixed edges from v to the nodes in A. To do so, fix some arbitrary time interval of T consecutive rounds. During the time interval, there are at most T rounds in which exactly one node transmits. Therefore, because $|A| = T+k$, there are at least k nodes in A which do not transmit alone during the given time interval. The edges from these k nodes to all nodes in B are therefore available throughout the T rounds. Therefore throughout any interval of T rounds, each node in B is connected to a set of at least k nodes in A. Consequently, the constructed dynamic network is T-interval k-connected. ◀

Notice that at least for store-and-forward algorithms even collision detection does not help to overcome the impossibility result. The 0-oblivious adaptive adversary knows the random choices of the algorithm in the current round and it can thus prevent any progress.

We also note that the above result turns out to be tight in the following sense. If $T \geq n-k$, global broadcast can be solved. If in each round, every node independently tries to broadcast with some probability (say $1/n$), if $T+k \geq n$ there is a non-zero probability (it may be very small) that T different nodes are isolated in T consecutive rounds. Consider an interval of T rounds and let I and U be the sets of informed and uninformed nodes at the beginning of this interval. From T-interval k-connectivity, we get that there are at least k nodes in I which are stably connected to nodes in I throughout the T rounds. Before broadcast is solved, we have $|I| \leq n-1$ and if in the T rounds, T different nodes in I are isolated, at least one of the k nodes stably connected to U gets isolated and we can therefore make progress. Note that for $T = n-k$, the probability for making progress might be exponentially small, resulting in an exponential running time for the broadcast problem. Note however also that once $T \geq cn \log n$ for a sufficiently large constant c, it is not hard to show that broadcast can be solved in polynomial time against a 0-oblivious adversary and if T is larger than $cn \log^2 n$ for a sufficiently large constant c, it is shown in [17], that it can be solved in time $O\left(n \log^2 n\right)$.

References

1 Mohamad Ahmadi, Abdolhamid Ghodselahi, Fabian Kuhn, and Anisur R. Molla. The cost of global broadcast in dynamic radio networks. *CoRR*, abs/1601.01912, 2016.

2 Antonio F. Anta, Alessia Milani, Miguel A. Mosteiro, and Shmuel Zaks. Opportunistic information dissemination in mobile ad-hoc networks: the profit of global synchrony. *Distributed Computing*, 25(4):279–296, 2012.

3 Chen Avin, Michal Koucký, and Zvi Lotker. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *Proc. 5th Coll. on Automata, Languages and Programming (ICALP)*, pages 121–132, 2008.

4 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. Efficient emulation of single-hop radio network with collision detection on multi-hop radio network with no collision detection. *Distributed Computing*, 5:67–71, 1991.

5 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.

6 Hervé Baumann, Pierluigi Crescenzi, and Pierre Fraigniaud. Parsimonious flooding in dynamic graphs. In *Proc. of 28th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 260–269, 2009.

7 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *Proc. 33rd Symp. on Principles of Distributed Computing (PODC)*, 2014.

8 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. A new perspective on vertex connectivity. In *Proc. 25th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 546–561, 2014.

9 Keren Censor-Hillel, Seth Gilbert, Fabian Kuhn, Nancy Lynch, and Calvin Newport. Structuring unreliable radio networks. *Distributed Computing*, 27(1):1–19, 2014.

10 Imrich Chlamtac and Shay Kutten. On broadcasting in radio networks–problem analysis and protocol design. *IEEE Transactions on Communications*, 33(12):1240–1246, 1985.

11 Andrea Clementi, Angelo Monti, Francesco Pasquale, and Riccardo Silvestri. Broadcasting in dynamic radio networks. *J. Comput. Syst. Sci.*, 75(4):213–230, 2009.

12 Andrea Clementi, Angelo Monti, Francesco Pasquale, and Riccardo Silvestri. Optimal gossiping in geometric radio networks in the presence of dynamical faults. *Networks*, 59(3):289–298, 2012.

13 Mohsen Ghaffari, Nancy Lynch, and Calvin Newport. The cost of radio network broadcast for different models of unreliable links. In *Proc. 32nd Symp. on Principles of Distributed Computing (PODC)*, pages 345–354, 2013.

14 Piyush Gupta and Panganmala R. Kumar. The Capacity of Wireless Networks. *IEEE Transactions on Information Theory*, 46(2):388–404, 2000.

15 Tomasz Jurdzinski, Dariusz R. Kowalski, Michal Rozanski, and Grzegorz Stachowiak. On the impact of geometry on ad hoc communication in wireless networks. In *Proc. 33rd Symp. on Principles of Distributed Computing (PODC)*, pages 357–366, 2014.

16 Kyu-Han Kim and Kang G. Shin. On accurate measurement of link quality in multi-hop wireless mesh networks. In *Proc. Conf. on Mobile Computing and Networking (MOBICOM)*, pages 38–49, 2006.

17 Fabian Kuhn, Nancy Lynch, Calvin Newport, Rotem Oshman, and Andréa W. Richa. Broadcasting in unreliable radio networks. In *Proc. 29th Symp. on Principles of Distributed Computing (PODC)*, pages 336–345, 2010.

18 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proc. 42nd Symp. on Theory of Computing (STOC)*, pages 513–522, 2010.

19 Fabian Kuhn and Rotem Oshman. Dynamic Networks: Models and Algorithms. *ACM SIGACT News*, 42(1):82–96, 2011.

20 Eyal Kushilevitz and Yishay Mansour. An ω(d\log(n/d)) lower bound for broadcast in radio networks. *SIAM journal on Computing*, 27(3):702–712, 1998.

21 Thomas Moscibroda and Roger Wattenhofer. Maximal independent sets in radio networks. In *Proc. 24th Symp. on Principles of Distributed Computing (PODC)*, pages 148–157, 2005.

22 Thomas Moscibroda and Roger Wattenhofer. The complexity of connectivity in wireless networks. In *Proc. 25th Conf. on Computer Communications (INFOCOM)*, pages 1–13, 2006.

23 Calvin Newport. Radio network lower bounds made easy. In *Distributed Computing*, pages 258–272. 2014.

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

25 Regina O'Dell and Roger Wattenhofer. Information dissemination in highly dynamic graphs. In *Proc. of Workshop on Foundations of Mobile Computing (DIALM-POMC)*, pages 104–110, 2005.

26 Krishna Ramachandran, Irfan Sheriff, Elizabeth Belding, and Kevin Almeroth. Routing stability in static wireless mesh networks. In *Proc. Conf. on Passive and Active Network Measurment*, pages 73–82, 2007.

27 Kannan Srinivasan, Maria A. Kazandjieva, Saatvik Agarwal, and Philip Levis. The β-factor: Measuring wireless link burstiness. In *Proc. 6th Conf. on Embedded Networked Sensor System*, pages 29–42, 2008.

28 Mark D. Yarvis, Steven W. Conner, Lakshman Krishnamurthy, Jasmeet Chhabra, Brent Elliott, and Alan Mainwaring. Real-world experiences with an interactive ad hoc sensor network. In *Proc. Conf. of Parallel Processing*, pages 143–151, 2002.

Bounds for Blind Rate Adaptation*†

Seth Gilbert[1], **Calvin Newport**[2], **and Tonghe Wang**[3]

1 **Department of Computer Science, National University of Singapore, Singapore**
seth.gilbert@comp.nus.edu.sg

2 **Department of Computer Science, Georgetown University, Washington, USA**
cnewport@cs.georgetown.edu

3 **Department of Computer Science, Georgetown University, Washington, USA**
tw473@georgetown.edu

Abstract

A core challenge in wireless communication is choosing appropriate transmission rates for packets. This rate selection problem is well understood in the context of unicast communication from a sender to a known receiver that can reply with acknowledgments. The problem is more difficult, however, in the multicast scenario where a sender must communicate with a potentially large and changing group of receivers with varied link qualities. In such settings, it is inefficient to gather feedback, and achieving good performance for every receiver is complicated by the potential diversity of their link conditions. This paper tackles this problem from an algorithmic perspective: identifying near optimal strategies for selecting rates that guarantee every receiver achieves throughput within reasonable factors of the optimal capacity of its link to the sender. Our algorithms have the added benefit that they are *blind*: they assume the sender has no information about the network and receives no feedback on its transmissions. We then prove new lower bounds on the fundamental difficulty of achieving good performance in the presence of fast fading (rapid and frequent changes to link quality), and conclude by studying strategies for achieving good throughput over multiple hops. We argue that the implementation of our algorithms should be easy because of the feature of being blind (it is independent to the network structure and the quality of links, so it's robust to changes). Our theoretical framework yields many new open problems within this important general topic of distributed transmission rate selection.

1998 ACM Subject Classification C.2.1 Network Architecture and Design, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases bitrate, multicast, packet transmission, latency, competitive ratio, lower bound, fading

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.8

1 Introduction

Consider the following scenario: a base station must wirelessly deliver a large file or stream a video to an unknown group of receivers in a conference center. It could send the data individually to each receiver using unicast communication, but this approach does not scale and requires knowledge of the group members. The standard alternative is for the sender

[*] Due to the matter of space, we defer details of the proof for some theorems or lemmas to the full version of this paper [8].

[†] The research in this paper has been supported by Singapore NUS FRC T1-251RES1404 and NSF grant number CCF-1320279.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 8; pp. 8:1–8:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to *multicast* messages to receivers, i.e., use one-to-many communication where the sender transmits packets addressed to the whole group.

A challenge faced by wireless multicast is that different receivers might have different quality links with the sender. Some receivers, for example, might have high quality links with the sender that can support high transmission rates, while others might have low quality links that can support only slow rates. *What rate(s) should the sender use?* The current answer – i.e., the solution implemented into 802.11 multicast [17] – is to transmit the multicast packets at the slowest rate. This solution is simple and reliable, but from a performance perspective it might be unacceptable for receivers capable of supporting much faster communication. Not surprisingly, practitioners consider the identification of better multicast rate selection strategies as an important open problem [10, 4, 16, 23, 3, 19, 6, 5, 22, 24].

In this paper, we tackle this open problem from an algorithmic perspective. We first formalize this rate selection problem in an abstract model of multi-rate wireless transmission. We then describe and analyze new rate selection strategies that guarantee *every* receiver in our above scenario achieves throughput within a reasonable factor of the optimal capacity of its sender link. We establish lower bounds indicating that these algorithms are near optimal. To the best of our knowledge, these are the first known rate selection strategies to offer competitive performance for every receiver in a wireless multicast scenario (see the related work below for more details). An added and perhaps surprising benefit of our algorithms is that they are also *blind*: the sender requires no knowledge of the network and receives no feedback on the fate of its transmission. This should make the implementation based on the algorithms robust to change.

1.1 Results

To formalize this multicast problem, assume a sender s and multiple receivers. For each packet to send, the sender must specify a transmission rate from a set of available rates. We normalize the transmission times associated with these rates such that the fastest rate delivers a packet in 1 time step, while the slowest delivers a packet in L time steps, for some system parameter $L > 1$. Notice, the range of possible rates with which a packet can be sent are fixed and provided by the radio hardware in most systems. Each receiver u is connected to s by a link labeled with a *fastest acceptable rate*. The sender succeeds in delivering a packet p to u iff it sends the packet at a rate no faster than the fastest acceptable rate for this link. Because we model a wireless network, the sender's packets are transmitted by broadcast. Therefore, each packet p, sent with some rate r, is received by *every* receiver with a link labeled by a fastest acceptable rate at least r. We measure the performance of the sender's rate selection strategy at receiver u by comparing the average latency between packets successfully delivered to u to the latency achieved if the sender had deployed the optimal strategy (for u) of transmitting every packet at the fastest acceptable rate for u's link. The sender's task is complicated by the fact that it must remain competitive for every receiver concurrently, even though their link qualities might vary widely and it receives no information on these qualities.

We begin by describing and analyzing a pair of blind rate selection algorithms (one randomized and one deterministic) that both guarantee that for each receiver v in the network, the throughput at v is within a $O(\log L)$-factor of optimal. Notice that the simple strategy of transmitting at the slowest available rate can be exponentially worse (i.e., achieve only an L factor of optimal). The core idea leveraged by both algorithms is to have the sender copy each packet into multiple queues, each associated with a different representative rate. The sender then dequeues and transmits messages from these queues at a frequency proportional to the corresponding rates (e.g., the fast rate queues are sampled more frequently than the slow rate queues). This means that each packet might end up being sent multiple times, but we show our proportional sampling prevents this from degrading throughput too much over time. We then establish this $O(\log L)$ competitive ratio near optimal by proving that no (randomized) blind rate selection algorithm can guarantee throughput better than a $\Omega(\frac{\log L}{\log\log L})$-factor of optimal (with constant probability).

We next turn our attention to the setting where the fastest acceptable rates on the links change rapidly and unpredictably, as might be described in a *fast fading* scenario. We prove that for every *deterministic* blind rate selection algorithm there exists a sequence of fades (i.e., link quality changes) that reduce its average latency guarantee to a trivial $\Omega(L)$-factor of optimal (which can always be matched within a constant factor by simply sending packets at the slowest rate). We then describe a type of fade for which no *randomized* algorithm can perform guarantee better than a $\Omega(\sqrt{L})$-factor of optimal (still exponentially worse than our results for the static setting). Interestingly, this latter bound holds even for non-blind unicast communication (i.e., where there is a single receiver and the sender learns the fate of each transmission), proving the difficulties caused by fading are not unique to blind algorithms.

To conclude, we consider multihop networks. We describe and analyze a generalization of the deterministic protocol that guarantees every receiver achieves throughput within a $O(\log L)$-factor of the optimal achievable through the best *single* multihop route (i.e., path) from the source. Notice, however, in a multihop network, the optimal throughput possible using *multiple* paths in the network might be better than the optimal throughput on a single path (e.g., perhaps multiple packets are routed concurrently on disjoint routes). We prove that no blind algorithm can guarantee a non-trivial approximation of this notion of optimal while also maintaining a fixed bound on its packet ordering.

1.2 Related Work

The multicast problem is well-studied in the wired network setting; c.f., [10]. In the wireless setting, the technology is still evolving. As mentioned, the default strategy implemented in 802.11 is to simply broadcast multicast packets at a slow but reliable bitrate. The research literature contains many proposals for adding more advanced functionality to wireless multicast, with a focus on detecting multicast packet loss. These strategies, however, depend on the sender interacting with at least some members of the multicast group (i.e., use feedback). Chandra et al. [4], for example, send the multicast data in unicast packets to a single member of the multicast group (leveraging link layer acknowledgments to detect packet loss), while the other members listen for these packets in promiscuous mode. Miroll et al. [16] refine this approach to select the group member with the worst channel as the unicast receiver to ensure more losses are detected. Sun et al. [23] organize nodes into clusters, and has the sender poll the leaders of each cluster to determine the fate of packets. (These are just a few examples among many: see [3] for a detailed survey.) The main goal of the above examples is to detect packet loss so the sender can schedule retransmissions. Most of these strategies, however, also implement some basic link adaptation. For example, when transmitting unicast packets to a leader, some of these strategies allow the default unicast rate adaptation strategy to adjust the rate used (e.g., [19, 6, 5, 22, 24]). Other wireless multicast strategies propose measuring loss rates for all receivers and using this information to choose the best *single* rate to use (as mentioned, a strategy that does not scale). This paper, by contrast, focuses on the problem of transmitting packets at multiple rates so as to ensure *every* receiver achieves a throughput close to its individual notion of optimal. It achieves this goal without the overhead and scaling issues of requiring feedback from group members.

It is also important to note that because we achieve competitive rates for every receiver, loss detection is less important with our scheme. For example, if we define the *fastest acceptable bitrate* for each receiver to be the fastest rate at which the forward error correcting coding parameters used for the packets are effective, then our blind rate selection algorithms will guarantee that every receiver has a sufficiently low loss rate to enable sufficient packet recovery.

Finally, wireless rate adaptation is well-studied in the context of unicast communication from a sender to a single known receiver capable of sending acknowledgments. Some strategies adapt the rate using frame loss information [12, 14, 11, 21, 15] and others attempt to directly measure the channel quality [18, 13, 25, 2]. Another approach is the use of rateless modulation schemes like Spinal codes [20] or Strider codes [9], in which the data transmitted in a fixed manner but the receiver is able to decode it at a rate close to the Shannon capacity for its channel. All these unicast adaptation strategies, however, depend on feedback from the single receiver to the sender. They cannot therefore be directly applied to the multicast scenario where such feedback is no longer efficient. Our blind protocols, by contrast, *can* be deployed in the unicast scenario. This might be desirable in low power scenarios where their simplicity provides an advantage, or scenarios such as satellite broadcast where feedback is prohibitively expensive (i.e., due to the much higher cost of uplink versus downlink transmission).

2 Model

We model a collection of wireless devices broadcasting in a synchronous radio network with variable link quality and transmission rates (typically called *bitrates* in the wireless literature as well as in the remainder of this paper). The network topology is represented as a connected directed graph $D = (V, E)$, where the nodes in V correspond to wireless devices and the edges in E represent links between nodes (e.g., an edge $(x, y) \in E$ means that x has a link to y with a quality above some minimum threshold). We use directed graphs for generality and to capture the well-known observation that link quality is not necessarily symmetric. The main topology we consider is a star with the node in the center playing the role of the sender with directed edges pointing toward the receivers. We call this configuration a *single hop* network. Later in the paper, we also examine the performance in general *multihop* networks of varying topologies connected with respect to the source (i.e., there is a path from the source to all nodes).

To capture link quality we assume that in each round, each *link* (i.e., edge in D) is assigned a minimum latency (i.e., fastest acceptable bitrate), which is specified by the weight function $C(r, e)$ (which we sometimes call a *channel*) for round r and edge e. We say that links are *static* if the weight of links does not change (i.e., $C(r, e) = C(r', e)$, for all r, r' and e); otherwise, we say that the links are *fading.* These weights capture the fastest transmission speed that can be supported by the current link quality. In this paper, we typically specify these weights in terms of the latency (i.e., rounds per packet), rather than in terms of the bitrate (which describes the inverse). Therefore, smaller weights represent higher quality links. We assume that all latencies are integers in $[L]$ (where we define $[k] = \{1, 2, ..., k\}$), with 1 and L rounds being the fastest and slowest transmission latencies, respectively. We also assume that L is a power of 2. To simplify our strategies, we will restrict the possible latencies considered for packet transmission to the set $\mathcal{L}^* = \{2, 4, 8, \ldots, L\}$. (Notice, there is a latency in $\mathcal{L}^*$ within a factor of 2 of each available latency.)

Nodes communicate with their neighbors in D using local broadcast. When a node s in D decides to broadcast a message in round r, it selects a latency $\ell \in \mathcal{L}^*$ (this is equivalent to selecting bitrate ℓ^{-1}). This transmission requires ℓ rounds to complete, and the transmitter must wait until the transmission completes before it can begin another transmission. Each neighbor w of s receives the message if the attempted broadcast latency remains as least as large as the minimal acceptable latency for the link throughout the full transmission. Formally, the transmission succeeds at w if and only if: $\forall i \in [0 \ldots \ell - 1] \; : \; C(r+i, (s,w)) \leq \ell$. If this condition does not hold, then node w does not receive the message. We assume no feedback mechanism (e.g., link layer ACKs) for the sender to learn the fate of its transmission, and assume nodes have no information about the network size or link weights.

A subtlety of our model motivation is that our abstract notion of "receiving" a message does not correspond to the concrete notion of a packet being successfully delivered. For u to "receive" a message from s in our model simply means that s sent this packet at a rate that was acceptable for its current link to u. What it means for a rate to be acceptable are details we abstract away: we aim only for every packet to be sent at acceptable rates for each receiver, for whatever definition of acceptable is relevant to a given a scenario.[1]

3 Problem

In this paper, we study the problem of a single distinguished *source* node s attempting to transmit an infinite stream of packets to the other nodes (called *receivers*) in the network. We measure the performance of an algorithm in a given execution by comparing the average packet receive latency at each receiver u with an offline optimal algorithm that services only u. We consider a restricted type of solution called a *blind rate selection algorithm*. An algorithm of this type running on the source node in the network is provided access to a packet queue called the *source queue*. To simplify definitions we assume the queue is infinite and the packets unique. The source node can only dequeue and transmit packets from the source queue (i.e., it cannot send arbitrary packets). In the multihop setting, where non-source nodes can forward packets, we assume each arriving packet is queued in a FIFO queue (ignoring duplicates), and then restrict nodes to dequeueing and transmitting packets from their local queue. In the single hop setting, receivers are passive (i.e., they cannot send packets.)

Recall, as described in the previous section, to "receive" a message in our abstract model simply means it was sent at an acceptable rate for the relevant link. How this translates to low level packet loss behavior is abstracted away. In this paper, we study the performance of correct blind rate selection algorithms defined with respect to the average time between packet arrivals at a given destination. We call this metric *average latency* (which is a mild abuse of terminology as "latency" often refers to end-to-end delivery, not inter-packet delay at a receiver). More precisely: Fix an execution of a blind rate selection algorithm in a network $D = (V, E)$ with weight function C. Fix some receiver $v \in V$ (i.e., non-source node) and integer duration $T \geq 1$. Let N_v^T be the number of unique packets received by v in the first T rounds of the execution. We define the *average latency* of v through the first T rounds of this execution to be T/N_v^T. By contrast, let OPT_v^T be the *optimal* average latency v could have achieved in these T rounds given an offline optimal schedule for transmissions and rate selections (when clear, we will use simply OPT). We now pull together these pieces to obtain the main performance definition:

[1] For example, an acceptable rate for a link in a given scenario might be defined as a rate for which the packet loss rate is sufficiently low. To send a packet at an acceptable rate in this example, therefore, does not guarantee that it was delivered, but merely that it was given a reasonable chance of delivery.

▶ **Definition 1.** Fix some function $f : \mathbb{N}^* \to \mathbb{R}$. We say a deterministic (randomized) blind rate selection algorithm $\mathcal{A}$ is $f(L)$-competitive with respect to a family $\mathcal{D}$ of networks and static/fading channels, if the algorithm is correct and there exists some integer duration $T_0 \geq 1$ such that for every $D \in \mathcal{D}$ and static/fading link weight function C, for every receiver v in D, and for every duration $T \geq T_0$: the (expected) average latency of v through T rounds in an execution of $\mathcal{A}$ in D with C, is no more than $f(L) \cdot OPT_v^T$.

4 Static Links

In this section, we study blind rate selection algorithms in the context of single hop networks with *static* links (i.e., the link qualities do not change). We begin by describing a simple randomized blind rate adaption algorithm that is $O(\log L)$-competitive in single hop networks. We then describe a more complex deterministic algorithm that matches this same $O(\log L)$-competitive ratio. There are two motivations for deterministic solutions. The first is that rate selection algorithms are often implemented at low layers of the network stack where efficient access to randomness is difficult. Second, the multihop algorithm studied later in the paper uses the deterministic algorithm as a key building block (analyzing the randomize strategy over multiple hops is difficult). We conclude this section by proving our algorithms near optimal with an $\Omega\left(\log L/\log\log L\right)$ lower bound on competitive ratio for blind rate adaption algorithms.

4.1 The RandSelect Algorithm

We model the infinite packet queue at the source s with the notation $Q_0 = p_1p_2p_3\ldots$. Recall, as defined in Section 2, $\mathcal{L}^* = \{2, 4, \ldots, L\}$.

A simple random strategy for s would be to dequeue packets from Q_0 one by one, sending each at a latency chosen uniformly from $\mathcal{L}^*$. An issue with this approach is reliability: if s chooses some latency ℓ for a given packet p_i, and there is some receiver u such that $C(s, u) > \ell$, then u will fail to receive p_i. Another issue with this strategy is that the slowest latency, L, will be chosen approximately once every $\log L$ rounds – requiring L full rounds for a single transmission every time it is chosen. This will yield non-competitive performance for receivers connected to the source with low latency links. Our proposed algorithm improves this simple scheme with two modifications to circumvent these two issues. First, the source maintains $\log L$ copies of its source queue, associating one copy with each latency in $\mathcal{L}^*$. We logically organize these queue copies into a *packet table* with one row for each latency. In more detail, each row $j \in [\log L]$ is associated with latency 2^j, and contains its own copy of the source queue, denoted Q_j, as well as a $nextpacket_j$ field which indicates the packet currently at the head Q_j. The second modification is to replace the uniform distribution over rates in $\mathcal{L}^*$ with the following distribution $\pi(x)$ over the latency indices $\{1, 2, ..., \log L\}$[2]:

$$\pi(x) : Pr\{j = x\} \begin{cases} 2^{-x} & x \in [1, \log L - 1] \\ 2/L & x = \log L \end{cases} .$$

Combining these modifications, our algorithm, which we call RandSelect, works as follows: At the beginning of the execution, the sender initializes its packet table by setting Q_j to Q_0 for each $j \in [\log L]$. It then proceeds by repeating the following steps: draw a *latency index* x from π, transmit $nextpacket_x$ at latency 2^x, and then update $nextpacket_x$

[2] Notice, it is easy to show that distribution π is normalized, i.e. $\sum_{x=1}^{\log L} Pr\{j = x\} = 1$.

```
RandSelect (for sender $s$)
Initialization:
  for $j \leftarrow 1$ to $\log L$
    $Q_j \leftarrow Q_0$
    $nextpacket_j \leftarrow p_1$

Transmission:
  do
    Select $j$ according to distribution $\pi$
    Send $nextpacket_j$ with latency $\ell \leftarrow 2^j$
    $pop(Q_j)$
    $nextpacket_j \leftarrow peek(Q_j)$
  while TRUE
```

Figure 1 The RandSelect Algorithm.

by dequeueing the packet at the head of Q_x. (See Figure 1 for the algorithm pseudocode.) Notice that this strategy overcomes both the issues described above for the simple random strategy: no packet is ever lost, as every packet is eventually sent at the slowest latency, and the algorithm now samples the slow latencies less frequently than the fast latencies, preventing them from dominating the link bandwidths.

The following theorem establishes that the actual competitive ratio guaranteed by this strategy is bounded by $O(\log L)$. We defer the proof of this theorem to the full version [8].

▶ **Theorem 2.** *The* RandSelect *blind rate selection algorithm is $O(\log L)$-competitive with respect to single hop networks and static links.*

A straightforward practical optimization would be to remove a packet from fast latency queues in the case that it is sent first by a slower latency. This optimization does not effect the asymptotic analysis.

4.2 The BCSSelect Algorithm

We now describe a deterministic blind rate selection algorithm we call BCSSelect (see Figure 2), which we will prove to have the same competitive ratio as RandSelect in single hop networks. The only difference between these two algorithm is how indices are selected. Our main strategy for derandomizing RandSelect is to leverage a useful object from number theory called the *binary carry seqeuence* (BCS) [1]. This BCS is defined such that its k^{th} term is the lowest position of a 1 bit in the binary representation of k. To use this sequence for our algorithms, we use the deterministic *schedule* function defined as follows: for $k \in \mathbb{N}^* : schedule(k) = \max\{\alpha \in \mathbb{N} : 2^{\alpha-1} | k\}$. The output of *schedule*, for example, produces the sequence: 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1... Before proceeding to our algorithm description (which uses the *schedule* output to sample queues), we first state some useful facts about *schedule* (first identified in [7] and reworded here):

▶ **Lemma 3.** *Each value $j \in \mathbb{N}$ is selected every 2^j iterations.*

▶ **Lemma 4.** *If $s = schedule(k)$, then:*
(i) $\forall t < s$, $\exists r \in [k - 2^{s-2}, k)$ *such that* $t = schedule(r)$*;*
(ii) $\forall t < s$, $\exists r \in (k, k + 2^{s-2}]$ *such that* $t = schedule(r)$*.*

```
BCSSelect (for source $s$)
Initialization:
  $k \leftarrow 1$
  for $j \leftarrow 1$ to $\log L$
    $Q_j \leftarrow Q_0$
    $nextpacket_j \leftarrow p_1$

Transmission:
  do
    $j \leftarrow schedule(k)$
    Send $(nextpacket_j, k)$ with latency $\ell = 2^j$
    $pop(Q_j)$
    $nextpacket_j \leftarrow peek(Q_j)$
    $k \leftarrow$ Update$(k)$
  while TRUE

Update$(k)$
  if $k = L/2$ then
    return 1
  else
    return $k + 1$
```

Figure 2 The BCSSelect Algorithm.

The BCSSelect algorithm (described in Figure 2), applies the *schedule* function as a subroutine to sample latencies from a (bounded) binary carry sequence – which ensures, as with the random distribution from before, that all latencies are sampled, but small latencies are sampled more often than their slower counterparts. In more detail, the algorithm uses *schedule* to sample the BCS until the first time latency L is sampled, at which point it restarts the sequence. As a result, the sequence of latencies sampled by *schedule* can be seen as repeating the same bounded BCS block with $L/2$ terms.[3] Here the source sends current BCS index k along with the packet, as this will prove useful in the later multihop version of the algorithm we study later.

We now argue that BCSSelect is $O(\log L)$-competitive, the same competitive ratio as RandSelect.

▶ **Theorem 5.** *The* BCSSelect *blind rate selection algorithm is $O(\log L)$-competitive with respect to single hop networks and static links.*

Proof. Fix receiver v with $C(s, v) = c$, where $\log c \in J = [1, \log L]$. Fix $T_0 = L \log L$ as well and suppose all the execution runs for $T \geq T_0$ rounds. By the same token, we have $OPT = c$ because the optimal solution runs with the sender v knowing $C(s, v) = c$ *a priori* and applying latency c throughout the execution.

Now the task is to find the average latency of BCSSelect. Consider the number of rounds required by the update of $nextpacket_{\log c}$, upper bounding the average latency we are looking for. One BCS block has $L/2$ terms because the greatest BCS index $j = \log L$ is

[3] Notice that the probability of selecting latency ℓ given by distribution π from RandSelect is equal to the proportion of latency ℓ in one such bounded BCS block.

selected for the first time when $k = L/2$. According to Lemma 3 and 4, the total time cost to generate one block is exactly the time needed to get another $j = \log L$ after the previous $j = \log L$, given by: $\sum_{j=1}^{\log L-1}(L/2)\cdot(1/2^j)\cdot 2^j + 2^{\log L} = L(\log L+1)/2$. Lemma 3 tells us that latency c (BCS index $\log c$) appears every $2^{\log c} = c$ iterations of *schedule*. So $\log c$ appears $L/2c$ times in one block, and the average time needed for $nextpacket_{\log c}$ to update will be $O\left(\frac{L(\log L+1)/2}{L/2c}\right) = O(c\log L)$.

In conclusion, the competitive ratio of BCSSelect during one block is $O(\log L)$. Because of the fact that the infinite latency sequence is actually the repeating of the same block, the competitive ratio during the whole execution process is still $O(\log L)$. ◀

4.3 Lower Bound

We now prove our algorithms near optimal by showing that every blind rate selection algorithm is at best $\Omega(\log L/\log\log L)$-competitive. Our argument is combinatorial in nature – it demonstrates that no sequence of rate selections can be sufficiently competitive for every receiver in a particular lower bound network – and therefore it applies to randomized solutions as well as deterministic.

▶ **Theorem 6.** *For a randomized blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to any single hop network and static links, then $f(L) \in \Omega(\log L/\log\log L)$.*

Assume for contradiction that we have some algorithm $\mathcal{A}$ that is $o(\log L/\log\log L)$-competitive for all networks and weight functions. For our lower bound, we define the following single hop *lower bound network*: let D be a directed graph that consists of $n = \log L/\log\log L$ receivers[4] denoted $r_1, r_2, ..., r_n$. Next, we define the weight function C such that for each $i \in [n]$, $C(s, r_i) = \log^i L$. That is, the weights in this network are: $W = \{\log L, \log^2 L, \log^3 L, ..., \log^{\log L/\log\log L} L = L\}$. We proceed with a series of proof step that build toward the conclusion that executing $\mathcal{A}$ in the lower bound network cannot yield the assumed small latency at every receiver. Our first step is to transform the algorithm $\mathcal{A}$ into an algorithm $\mathcal{B}$ that uses only the latencies in W. The following lemma states that there exists transformation of this type that do not affect performance. The proof is deferred to the full version of this paper [8].

▶ **Lemma 7.** *Let $\mathcal{A}$ be a rate adaptation algorithm that is $f(L)$-competitive in the lower bound network. There exists an algorithm $\mathcal{B}$ that only selects latency in W but is still $f(L)$-competitive in the lower bound network D.*

Fix some blind rate selection algorithm $\mathcal{B}$ that only uses latencies in W. We now prove that a constant fraction of the packets received by a given receiver must be at a "good" rate (i.e., the link weight) for that receiver.

▶ **Lemma 8.** *Let $\mathcal{B}$ be a rate adaptation algorithm that only selects latencies in W and is $f(L)$-competitive in the lower bound network D. Fix some duration $T_0 = L$. Let k_i be the number of messages received by receiver r_i in a T round execution of $\mathcal{B}$, where $T \geq T_0$. It follows that at least $\lceil k_i/2\rceil$ of these messages are sent at latency $\ell_i = \log^i L$.*

Proof. Assume for contradiction that half or more of these k_i message were sent at a latency greater than ℓ_i. The minimum latency for a packet sent to receiver r_i is ℓ_i. Therefore, if we

[4] For notational simplicity we assume $\log L$ and $\log L/\log\log L$ are positive, integral values.

assume that at least half the messages arriving at r_i are sent at a latency longer than ℓ_i, the next best case average latency for r_i would be at least: $((k/2)\ell_{i+1} + (k/2)\ell_i)/k$.

In the above, we assume the minimum number of packets (in this case, $k/2$) were sent at a slower latency, and we made this the next slowest latency after ℓ_i (i.e., $\ell_{i+1} = \log^{i+1} L$). The rate above also assumes the source was only servicing r_i and sent packets continuously with no gaps throughout the T rounds. It is, in other words, a quite optimistic bound. We now simplify:

$$\frac{(k/2)\ell_{i+1} + (k/2)\ell_i}{k} = \frac{(k/2)\ell_i \log L + (k/2)\ell_i}{k} = \frac{(1/2) \cdot k \cdot \ell_i(\log L + 1)}{k} > \frac{1}{2} \cdot \ell_i \cdot \log L\,.$$

The optimal average latency for r_i is clearly ℓ_i. Therefore, $\mathcal{B}$ is *at best* $(\log L/2)$-competitive. We assumed earlier, however, that $\mathcal{B}$ is $f(L)$-competitive for a function that is no larger than $c \cdot \log L/\log\log L < \log L/2$ (for sufficiently small constant $c > 0$). This contradicts our assumption that at least half of r_i's packets were sent slowly. ◀

We have just established that to achieve a reasonable competitive ratio for a given receiver in our network, at least half of the packets sent to the the receiver must use a rate well-suited to the receiver's link. We next establish formally another important observation for our overall lower bound: to achieve a good rate in an execution of length T, the source must successfully deliver many packets.

▶ **Lemma 9.** *Let $\mathcal{B}$ be an algorithm that is $f(L)$-competitive in the lower bound network D. Fix some receiver r_i and duration $T \geq T_0 = L$. It follows that r_i receives at least $T/(\ell_i \cdot f(L))$ packets during these T slots, where $\ell_i = \log^i L$.*

Proof. Fix some $f(L)$-competitive algorithm $\mathcal{B}$, as well as some r_i and $T \geq T_0 = L$. Consider a T-round execution of $\mathcal{B}$ in the lower bound network. Let $\ell_1^*, \ell_2^*, ..., \ell_j^*$ be the latency for each receive event at r_i in our T-round interval. Let α be the average latency at r_i in this interval. Notice, by definition: $\alpha = (1/j) \cdot \sum_{h=1}^{j} \ell_h^*$. Also note, however, that by definition: $\sum_{h=1}^{j} \ell_h^* = T$. It follows that $\alpha = T/j$, and therefore $j = T/\alpha$. By assumption, $\mathcal{B}$ is $f(L)$-competitive. This means that when considering r_i in particular, its average latency is no greater than $\ell_i \cdot f(L)$ and $j \geq T/(\ell_i \cdot f(L))$, as required. ◀

Proof (of Theorem 6). Let $\mathcal{A}$ be the rate adaptation algorithm that we assumed to be $f(L)$-competitive for some $f(L) < c \cdot \log L/\log\log L$. Let $\mathcal{B}$ be the constrained rate adaptation algorithm provided by Lemma 7. By the guarantees of this lemma, $\mathcal{B}$ is $f(L)$-competitive in the lower bound network D. We will now show that this leads to a contradiction.

In particular, we will show that for any sufficiently large duration $T \geq T_0 = L$, $\mathcal{B}$ is at best $\Omega(\log L)$ competitive which is $\omega(f(L))$ which contradicts our assumption that it is $f(L)$-competitive.

To get this result, let k_i be number of packets that the source sends at latency $\ell_i \in W$ in a T-round execution of $\mathcal{B}$ in the lower bound network D. By Lemma 9, we know that receiver r_i receives at least $T/(\ell_i \cdot f(L))$ packets. By Lemma 8, we know at least half these packets are sent at latency ℓ_i. It follows that $k_i \geq T/(2 \cdot \ell_i \cdot f(L))$. We can now evaluate how many rounds are required for the source to send the needed number of packets at each rate, and derive the following answer:

$$\sum_{i=1}^{n} k_i \cdot \ell_i = \sum_{i=1}^{\log L/\log\log L} \frac{T}{2 \cdot \ell_i f(L)} \cdot \ell_i = \frac{\log L}{\log\log L} \cdot \frac{T}{2f(L)}\,.$$

By assumption, however, $f(L) < c \cdot \log L/\log\log L$. If we set c to be sufficiently small (e.g., $c < 1/2$), it simplifies to something strictly larger than T. There are only T rounds available, however, to complete all broadcasts. This yields a contradiction to our assumption about the bound on $f(L)$, and therefore $f(L)$ is in $\Omega(\log L/\log\log L)$ ◀

5 Fast Fading Links

In this section, we consider the setting where link weights can change from round to round. It is straightforward to identify a weight function C that causes our static BCSSelect algorithm to perform poorly in the face of some dynamism (i.e., achieve only an $O(L)$ competitive ratio with respect to optimal).

Here we generalize this observation by proving this weakness is true of *all* deterministic blind rate selection algorithms. We prove that there is a link weight definition for which *no* randomized algorithm can guarantee better than a $\sqrt{L}$-competitive ratio (which is still exponentially worse than the $\log L$ ratio we achieve for static links). Perhaps surprisingly, this latter bound even holds for the powerful model of unicast communication with a single receiver and packet feedback. These bounds indicate that it is quixotic to seek an algorithm that can always adapt competitively to fast fades.

Lower Bound for Deterministic Algorithms. A deterministic blind rate selection algorithm can be described as a fixed sequence of latency choices. Here we prove that for any such sequence we can define a link weight function for a two-node network (the simplest possible network for rate selection) that guarantees a poor competitive ratio.

▶ **Theorem 10.** *For a deterministic blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to two-node networks and fading links, then it follows that $f(L) \in \Omega(L)$.*

The proof of this theorem is deferred to the full version [8]. At a high-level, this argument defines a weight function that keeps the link weight large when the algorithm attempts fast transmissions, and reduces the weight to something small when the algorithm attempts slow transmissions.

A Lower Bound for Randomized Algorithms. Here we show that randomization cannot guarantee much advantage over determinism given fading links. The following theorem holds even for non-blind rate selection algorithms in which the sender learns the fate of each packet.

▶ **Theorem 11.** *For a randomized blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to two-node networks and fading links, then it follows that $f(L) \in \Omega(\sqrt{L})$. This bound holds even with packet delivery acknowledgements.*

The proof of Theorem 11 depends on the following lemma.

▶ **Lemma 12.** *For a blind rate selection algorithm $\mathcal{A}$, if $\mathcal{A}$ is $f(L)$-competitive with respect to two-node networks and fading links, then for every function $g : \mathbb{N}^* \to \mathbb{R}$ such that $g(L) < L/2$,*

$$f(L) \in \Omega\left(\min\left\{(L/g(L))\,, g(L)\right\}\right).$$

This bound holds even with packet delivery acknowledgements.

To come close to the optimal solution, a randomized algorithm must effectively guess correctly the beginning of the fast interval in each block. Receiving feedback after the fact regarding whether it guessed correctly does not help its future guesses. We formalize this argument in the full version of this paper [8]. Armed with this lemma, we can prove our theorem.

Proof (of Theorem 11). According to Lemma 12, for any appropriately chosen $g(L)$, the lower bound for any randomized blind rate adaption algorithm $\mathcal{A}$ is $\Omega\left(\min\left\{\left(L/g(L)\right), g(L)\right\}\right)$. In particular, the strongest lower bound $\Omega(\sqrt{L})$ is achieved when $g(L) = \sqrt{L}$, and the duration $T \geq T_0 = 2g(L) = 2\sqrt{L}$. ◀

6 Multihop Networks

In this section, we turn our attention to routing information through multihop networks. In particular, consider a multihop network in which the source s may not have a direct link to some designated receiver t. In this setting, s will have to forward messages through intermediate nodes to get to t, with each such node needing to make its own rate selection decisions.

We consider two natural methods to measure optimality with respect to t. The first method is to consider any single path from s to t in the network, and compare t's throughput when the algorithm is run on this path to the throughput obtained by the optimal algorithm for the path. We call this *single path* optimality. We describe a deterministic algorithm called MultiBCSSelect that generalizes the single-hop BCSSelect algorithm to obtain throughput within a $O(\log L)$-factor of the single path optimal solution. The second method is to compare the throughput at t when the algorithm is executed in the entire network as compared to the optimal algorithm executed in the entire network. Notice, once you make use of the entire network, it might be possible to obtain more performance (e.g, by routing multiple packets to the destination concurrently over disjoint paths). Our MultiBCSSelect algorithm cannot guarantee this *multiple path* optimality. We prove, however, that in some sense *no* blind algorithm can. In more detail, we prove that it is impossible for a blind rate selection algorithm to guarantee a non-trivial approximation of the multiple path optimal solution *and* to be δ-order preserving (i.e., sequence numbers of received packets do not get more than δ values out of order), for any fixed δ. We note that this latter property is necessary for many network applications, and our MultiBCSSelect algorithm is 0-order preserving.

6.1 The MultiBCSSelect Algorithm

Here we describe a blind rate adaptation algorithm for multihop packet transmission based on BCSSelect. In particular, we have the source node s run BCSSelect, as in the single hop setting. The non-source nodes, by contrast, run the MultiBCSSelect algorithm described in Figure 3. This algorithm initializes each Q_j at an intermediate node as an empty queue. As an intermediate node receives a packet for the first time, it pushes it onto the back of each of its queues. This algorithm has nodes sample queues as in the single hop algorithm. In the case that it samples an empty queue, the node will simply transmit an "empty packet" (technically, we can interpret this as not sending any packet). We synchronize the indexes nodes use to sample the BCS by propagating the current index in the transmitted packets.

It is straightforward to show that MultiBCSSelect is correct in a multihop setting. More interesting is analyzing its performance. Because we consider single path optimality, we restrict our attention to a subgraph P consisting of a path from s to some fixed destination t, i.e., $V_P = \{v_0 = s, v_1, v_2, \ldots, v_n, v_{n+1} = t\}, E_P = \{(v_i, v_{i+1}) : i = 0, 1, \ldots, n\}$. We show that

```
MultiBCSSelect(for intermediate nodes)
Initialization:
  for $j \leftarrow 1$ to $\log L$
    $Q_j$ is initialized by an empty packet queue

On receiving $(p_0, k_0)$:
  $k \leftarrow$ Update$(k_0)$
  for $j \leftarrow 1$ to $\log L$
    $Q_j \leftarrow push(Q_j, p_0)$
    $nextpacket_j \leftarrow peek(Q_j)$

Transmission:
  do
    $j \leftarrow schedule(k)$
    Send $(nextpacket_j, k)$ with latency $\ell = 2^j$
    $pop(Q_j)$
    $nextpacket_j \leftarrow peek(Q_j)$
    $k \leftarrow$ Update$(k)$
  while TRUE
```

Figure 3 Algorithm of MultiBCSSelect.

the performance of MultiBCSSelect is competitive with that achieved by the best path P, i.e., the best multihop route to t. (The best multihop route is the path which gives the highest throughput.)

▶ **Theorem 13.** *The* MultiBCSSelect *blind rate selection algorithm is $O(\log L)$-competitive with respect to the single path optimal solution.*

Before completing the proof of Theorem 13, we will bound the performance of the optimal algorithm on P (the proof of this lemma is in [8]):

▶ **Lemma 14.** *Fix a multihop route consisting of the path P with $n+2$ nodes $v_0, \ldots, v_{n+1}$, where $s = v_0$ and $t = v_{n+1}$. Suppose $c^* = \max_{0 \le i \le n}\{C(v_i, v_{i+1})\}$. For all multihop routes with n intermediate nodes, if the links are static, the optimal average latency $OPT_t = \Omega(c^*)$.*

Proof (of Theorem 13). Now we need to capture the average latency of MultiBCSSelect. Consider some link $(v_\beta, v_{\beta+1})$ with β being the greatest index such that $C(v_\beta, v_{\beta+1}) = \max_{0 \le \alpha \le n}\{C(v_\alpha, v_{\alpha+1})\} = c^*$. In other words, $(v_\beta, v_{\beta+1})$ is the last bottleneck, or the last slowest link. Since MultiBCSSelect applies synchronous binary carry sequence, a packet will be in transmission successfully before any packet arrives, indicating that v_β is the last place where packets get queued.

When running algorithm MultiBCSSelect, according to the analysis from Theorem 5, the link with weight c sends a new packet within $c \log L$ rounds during one block of binary carry sequence. We will see that the worst case for transmission through path P derives from the case where $C(v_i, v_{i+1}) = c^*$ for all $i > \beta$. Then the average time for v_n to update $nextpacket_{\log c^*}$ is no more than $O(c^* \log L)$, and the corresponding competitive ratio on this path is therefore $O(\log L)$.

Since this is true for all paths P, including the best such path, we have proved our claim. ◀

6.2 Lower Bound for Multiple Path Optimality

Here we show it is impossible to be multiple path optimal and still maintain a natural packet ordering property. This latter property is captured by the following two definitions.

▶ **Definition 15.** We define the sequence number of a packet p, denoted by $seq(p)$, to be the order of packet p in the source's packet queue at the beginning of the execution (where the packet at the head of the queue occupies position 1, and so on). Fix some non-source node t. Similarly, we define the transmission number of p with respect to t, denoted by $tn(t, p)$, to be the order in which t first received p, ignoring duplicate receives of packets.

▶ **Definition 16.** Fix some integer $\delta \geq 0$. A rate adaptation algorithm is δ-order preserving if for every packet p in the source queue, and every non-source node t, we have $|seq(p) - tn(t, p)| \leq \delta$.

This notion of δ-order preserving is important for many applications in which received packets need to be reordered for processing. If I need, for example, k out of an original group of t packets to recover some coded information, and some of these packets can get arbitrarily out of order, I might have to wait an arbitrarily long time to complete the decoding.

We continue by noting that our multihop algorithm is perfectly order preserving:

▶ **Theorem 17.** MultiBCSSelect *is* 0*-order preserving.*

Proof. The source node copies its source queue into $\log L$ transmission queues, each one associated with a different latency. Packets are removed and transmitted from each queue in FIFO order. A straightforward consequence is that for any two packets p and p', such that $seq(p) < seq(p')$, the source cannot send p' for the first time before it sends p (consider the queue from which the source samples p' for the first time: in order to reach p' in that queue, the source must have previously sampled and transmitted p).

It then follows that all neighbors of the source will receive messages for the first time in the same order as they appear in the source queue. They will subsequently add them to their transmission queues the same way. We can, therefore, apply the same argument as before to show this order is preserved to their neighbors, and so on, until we have considered every node in the network. It follows that for any non-source node t and any two packets p and p, if $seq(p) < seq(p')$, then $tn(t, p) < tn(t, p')$. ◀

With these definitions established, we can now state our main theorem, which claims that a blind algorithm cannot be both non-trivially competitive with respect to the multiple path optimal results, and be order preserving for some fixed δ.

▶ **Theorem 18.** *There exists a constant $c' > 1$, such that for every integer $\delta \geq 0$ and competitive factor $c < L/c'$, there does not exist a blind rate adaptation algorithm that is c-competitive with respect to the multiple path optimal path solution* and *δ-order preserving.*

To prove this lower bound, we will make use of a graph $D_r = (V, E)$, where V consists of a source, s, a destination, t, and L relay nodes, $r_1, r_2, \ldots, r_L$. Let $E = \{(s, r_i) : 1 \leq i \leq L\} \cup \{(r_i, t) : 1 \leq i \leq L\}$. Fix $C(s, r_i) = 1$ and $C(r_i, t) = L$ for all $i = 1, 2, \ldots, L$.

Fix some rate adaptation algorithm $\mathcal{A}$ that is c-competitive for some constant $c > 0$. To prove Theorem 18, we will show that there exists a network such that for all $x \geq 1$, there exists a packet p such that $|seq(p) - tn(t, p)| = \Omega(x \cdot L)$. For any fixed δ, therefore, we can find a sufficiently large x for which the algorithm is not δ-order preserving.

Let us study the constant competitive algorithm $\mathcal{A}$. Since all links coming out of the source have the same capacity, relay nodes will receive the same packet in the same transmission round. In order to achieve good competitiveness, relay node may not send packets in FIFO order. Otherwise, there will be a great amount of repetition of packets at the destination t, since relay nodes receives the same packet in each round. Actually we can claim without proof that the constant competitive solution for one single transmission round is to have L different relay nodes send $\Theta(L)$ different packets in the queue.

We will prove that after x transmission rounds (xL communication rounds), the maximum sequence number of the packets that have already been sent will be $\Omega(x) \cdot \Theta(L) = \Omega(xL)$ for all $x \geq 1$. We will forget about the first $L+1$ communication rounds and regard the start of the $(L+2)$th rounds as the start of $x = 1$.

▶ **Lemma 19.** *When executing $\mathcal{A}$ on graph D_r, there exists some relay node r, such that for every integer $x \geq 1$, there exists an integer $x' \geq x$, such that $seq(p_{x'}^{(r)}) \geq (1/c)x'L$, where $p_{x'}^{(r)}$ is the x'th unique packet that r sends.*

We will put the proof of this lemma in the full version [8]. A simple counting argument yields the next lemma:

▶ **Lemma 20.** *When executing $\mathcal{A}$ in any network, for every node u, after u transmits x unique packets, the smallest sequence number among packets u has not yet sent is no more than $x+1$.*

We can now pull together the pieces to prove our main theorem.

Proof (of Theorem 18). The key observation used by this proof is that a relay node r_i cannot distinguish an execution in D_r from an execution in the graph $D_r^{(i)}$ which consists only of: the source s, with a directed edge to r_i, with a directed edge to t. Now consider an execution of $\mathcal{A}$ in L copies of $D_r^{(i)}$, one for each r_i. At the same time, run this algorithm with the same random bits in D_r. We will look at the behavior of $\mathcal{A}$ in D_r to point to an i for which $D_r^{(i)}$ behaves poorly.

In more detail, we apply Lemma 19, which identifies some r_i in D_r for which the lemma statement holds. Let x be the value identified by the statement for r_i. Let $x' = \max\{x, (\delta+1)/(\frac{L}{c}-1)\}$, where δ is the order-preserving bound from the theorem statement. Consider $p_{x'}^{(i)}$, the x'th packet sent by r_i. By the statement, $seq(p) \geq (1/c)x'L$. By Lemma 20, however, there is some sequence number $q \leq x'+1$, such that r_i has not yet sent the packet with that number.

Now consider r_i in $D_r^{(i)}$. It too will send a packet with sequence number at least $(1/c)x'L$ before it sends a packet with number $x'+1$. Because r_i must eventually send every packet in this graph (as it is the only relay node), when it does eventually get to the packet with sequence number $x'+1$, it will be out of order. In particular, the gap between this packet's number and the x'th packet's number is at least: $x'(L/c) - q \geq x'(L/c) - (x'+1) > \delta$. (Notice, it is here that we require that c is sufficiently small compared to L.) We have just identified, however, an execution of $\mathcal{A}$ in a graph that is non-order preserving. A contradiction. ◀

7 Conclusion

In this paper, we study *blind* multicast rate selection algorithms which do not require feedback from receivers, and yet allows each receiver to achieve throughput within a reasonable constant factor of its link's optimal capacity. We prove these algorithms near optimal and then explore the fundamental impossibilities of coping with fast fading, even for non-blind algorithms. We conclude by showing how our deterministic strategy can be effectively adapted to multihop scenarios.

We argue that our algorithms are easy to implement in practice, because they are blind, or they does not require any information on the network structure or the quality of links. Our formal model of multi-rate transmission, however, is a standalone contribution as it helps bring together the practical concerns of rate selection with the theoretical toolkit of algorithmic analysis. This framework yields many interesting open questions. For example, this paper only scratches the surface of understanding optimal rate selection in general network topologies. Even identifying an efficient centralized solution for approximating an optimal selection sequence is an open problem. Another natural approach would be to consider unicast rate selection where the sender receives feedback on each transmitted packet's fate. The existing solutions for this problem rely on heuristics. It would be useful to study this problem from an algorithmic perspective, seeking formal bounds.

References

1 K. Atanassov. On the 37th and the 38th Smarandache problems. *Notes on Number Theory and Discrete Mathematics*, pages 83–85, 1999.

2 Saâd Biaz and Shaoen Wu. Loss differentiated rate adaptation in wireless networks. In *IEEE WCNC 2008*, 2008.

3 Saâd Biaz and Shaoen Wu. Rate adaptation algorithms for IEEE 802.11 networks: A survey and comparison. In *Proceedings of IEEE Symposium on Computers and Communications*, 2008.

4 R. Chandra, S. Karanth, T. Moscibroda, V. Navda, J. Padhye, R. Ramjee, and L. Ravindranath. Dircast: A practical and efficient Wi-Fi multicast system. In *Proceedings of the 17th IEEE International Conference on Network Protocols*, 2009.

5 N. Choi, Y. Seok, T. Kwon, and Y. Choi. Leader-based multicast service in IEEE 802.11v networks. In *Proceedings of the 7th IEEE Consumer Communications and Networking Conference*, 2010.

6 S. Choi, N. Choi, Y. Seok, and T. Kwon. Leader-based rate adaptive multicasting for wireless LANs. In *Proceedings of IEEE Global Telecommunications Conference*, 2007.

7 Alejandro Cornejo and Calvin Newport. Prioritized gossip in vehicular networks. In *DIALM-POMC'10*, 2010.

8 Seth Gilbert, Calvin Newport, and Tonghe Wang. Bounds for blind rate adaptation. Available at: `http://people.cs.georgetown.edu/~cnewport/publications.html`.

9 Aditya Gudipati and Sachin Katti. Stanford networked systems group. `http://snsg.stanford.edu/projects/strider/`.

10 Lawrence Harte. *Introduction to Data Multicasting.* Althos Publishing, 2008.

11 G. Holland, N. Vaidya, and P. Bahl. A rate-adaptive MAC protocol for multi-hop wireless networks. In *ACM MOBICOM'01*, 2001.

12 A. Kamerman and L. Monteban. WaveLAN II: A high-performance wireless LAN for the unlicensed band. *Bell Labs Technical Journal*, 1997.

13 J. Kim, S. Kim, S. Choi, and D. Qiao. CARA: Collision-aware rate adaptation for IEEE 802.11 WLANs. In *IEEE INFOCOM'06*, 2006.

14 M. Lacage, M. Manshaei, and T. Turletti. IEEE 802.11 rate adaptation: A practical approach. In *MSWiM04*, 2004.

15 Z. Li, A. Das, A. K. Gupta, and S. Nandi. Full ato rate MAC protocol for wireless ad hoc networks. In *IEEE Proceedings on Communication*, 2005.

16 J. Miroll and Z. Li. Aggregate block-ACK definition. *Tech. Rep. IEEE*, 2010.

17 Sai Shankar N., Debashis Dash, Hassan El Madi, and Guru Gopalakrishnan. WiGig and IEEE 802.11ad for Multi-Gigabyte-Per-Second WPAN and WLAN. *arXiv:1211.7356*, 2012.

18 Qixiang Pang, Victor Leung, and Soung C. Liew. A rate adaptation algorithm for IEEE 802.11 WLANs based on MAC-layer loss differentiation. In *Proceedings of IEEE Broadband Wireless networking symposium*, 2005.

19 Y. Park, Y. Seok, N. Choi, Y. Choi, and J.-M. Bonnin. Rate-adaptive multimedia multicasting over IEEE 802.11 wireless LANs. In *Proceedings of the 3rd IEEE Consumer Communications and Networking Conference*, 2006.

20 Janathan Perry, Hari Baladrishnan, and Devavrat Shah. Rateless spinal codes. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

21 B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly. Opportunistic media access for multirate ad hoc networks. In *MOBICOM02*, 2002.

22 Y. Seok and T. Turletti. Practical rate-adaptive multicast schemes for multimedia over IEEE 802.11 WLANs, 2006. https://hal.inria.fr/inria-00104699.

23 M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai. Reliable MAC layer multicast in IEEE 802.11 wireless networks. In *Proceedings of International Conference on Parallel Processing*, 2002.

24 J. Villalón, P. Cuenca, L. Orozoco-Barbosa, Y. Seok, and T. Turletti. Cross-layer architecture for adaptive video multicast streaming over multirate wireless LANs. *IEEE J. Sel. Areas Commun.*, 25(4):699–711, 2007.

25 S. Wong, H. Yang, S. Lu, and B. Bharghavan. Robust rate adaption for 802.11 wireless networks. In *MOBICOM'06*, 2006.

Overcoming Obstacles with Ants

Tobias Langner[1], Barbara Keller[2], Jara Uitto[3], and Roger Wattenhofer[4]

1 **ETH Zürich, Zürich, Switzerland**
`tobias.langner@tik.ee.ethz.ch`
2 **ETH Zürich, Zürich, Switzerland**
`barbara.keller@tik.ee.ethz.ch`
3 **ETH Zürich, Zürich, Switzerland**
`jara.uitto@tik.ee.ethz.ch`
4 **ETH Zürich, Zürich, Switzerland**
`roger.wattenhofer@tik.ee.ethz.ch`

Abstract

Consider a group of mobile finite automata, referred to as agents, located in the origin of an infinite grid. The grid is occupied by *obstacles*, i.e., sets of cells that can not be entered by the agents. In every step, an agent can sense the states of the co-located agents and is allowed to move to any neighboring cell of the grid not blocked by an obstacle. We assume that the circumference of each obstacle is finite but allow the number of obstacles to be unbounded. The task of the agents is to cooperatively find a treasure, hidden in the grid by an adversary.

In this work, we show how the agents can utilize their simple means of communication and their constant memory to systematically explore the grid and to locate the treasure in finite time. As integral part of the agents' behavior, we present a method that allows a group of six agents to follow a straight line, even if the line is partially obstructed by obstacles, and to discover all free cells along this line. In total, our search protocol requires nine agents.

1998 ACM Subject Classification F.1.1 Computation by Abstract Devices: Models of Computation

Keywords and phrases Mobile agents, algorithms, treasure search

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.9

1 Introduction

How do ants find that crumb of chocolate dropped on the kitchen floor? And how do they navigate through that huge Lego castle built by the children to get to the crumb? General knowledge is that such an amazing achievement can be explained by so-called pheromones, a chemical factor used by ants to mark the terrain. However, as it turns out, many ant species do not use pheromones at all, and instead communicate with their antennae when bumping into each other [21]. So how do they do it – are ants pretty intelligent after all?

In this paper, we model a single ant with a mobile version of a finite state machine and will later in the paper refer to an ant as an agent. If two ants meet, they can influence their states, no other form of communication is allowed. We show that a small group of nine of our ants will collaboratively be able to find a treasure in an arbitrarily obstructed environment. Our ants use only a constant amount of memory, independent of the distance from the nest to the treasure, and the number and size of the obstacles.

Our result is intended as proof of concept showing that it is indeed possible to search a grid with obstacles with mobile finite state machines. To do so, however, our protocol might

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 9; pp. 9:1–9:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

require the ants to walk around obstacles of arbitrary size, which forbids a runtime-bound independent of the obstacle size. It is an interesting open problem whether this dependency can be avoided or not. Our efforts suggest that it might not be possible.

2 Related Work

Recently, scientists in biology and computing have been flirting with each other. Distributed computing in particular seems to be a valuable tool towards understanding biological phenomena, as both often deal with networks of simple nodes, collaborating by means of minimal communication. Please see the recent survey from Navlahka and Bar-Joseph for more details [19].

Ants in particular have been a focus of interest in the Computer Science community. As an example, Feinerman et al. modeled the foraging behavior of ants as an exploration problem, where n agents are collaboratively searching the plane and the goal is to find an adversarially hidden treasure [13, 14]. In a similar setting, Lenzen et al. studied the effects on bounding the memory and the range of available probabilities for the agents [18]. Our model is a variant of their model, where the agents are controlled by finite state machines instead of Turing machines. Without obstacles, but allowing communication, it was shown that asymptotically there is no penalty when ants are restricted to finite state machines [12]. In the case of an infinite grid without obstacles, it was discovered by Emek et al. that two deterministic finite state machines cannot discover every cell [11]. In the same work, it was also shown that a randomized finite state machine requires infinite time in expectation and that four (deterministic) finite state machines are always enough to discover the treasure. Since the unobstructed infinite grid is a special case of our setting, the same lower bounds hold for our problem. However, it seems that introducing obstacles fundamentally changes the picture. In this paper we show that it is still possible to discover the treasure in this more challenging setting and we derive an upper bound for the number of ants required for it.

Our work also has connections to graph exploration, as the problem we are studying is a variant of it. In the general graph exploration setting, the goal is to visit all the nodes or all the edges of a graph starting from any node. In our work the unobstructed cells can be interpreted as nodes and their connections to their neighbors as edges. The task of the agents is to discover all unobstructed cells. Graph exploration has been extensively studied in the literature and the studies can be divided into two settings. One of the settings is to assume that the graphs are directed, i.e., an edge can only be traversed in one direction, not vice versa [1, 3, 7]. In the other, the edges can be traversed to both directions [2, 8, 10]. Our work belongs to the second setting, as the agents can move back and forth between neighboring cells.

Furthermore, there are two main types of performance measures regarding graph exploration. The first measure is the time complexity, i.e., how long does it take for the agent(s) to finish the exploration task [20]. The other one is to measure the bit complexity, i.e., how many bits of memory does the agent(s) require to solve the exploration task [15]. Furthermore, the aforementioned graph exploration tasks can be considered with return, stop, or perpetual properties, i.e., whether the agent is required to return to the starting cell, stop the search after finishing, or if the agent is not required to terminate [8, 16]. Note that even though in [16] a finite automaton explores a graph, this automaton is equipped with a memory linear in size of the diameter of the graph. In our work we show that our finite automata only need constant memory to solve the task.

Since we are restricting our underlying graph to $\mathbb{Z}^2$ and the obstacles in our domain essentially block the agents from entering specific cells, our graphs correspond to a concept widely studied in literature called *labyrinths* [4, 9]. Exploration of a labyrinth corresponds to the task of getting as far from the starting point as possible, for any starting point. It was shown by Budach that a single automaton cannot explore every finite labyrinth, where a finite labyrinth has only a finite amount of blocked cells [6]. On the positive side, it is known that every finite labyrinth can be explored by a finite automaton using four pebbles and that all co-finite (number of non-blocked cells is finite) labyrinths can be explored with a finite state machine using two pebbles [5]. A pebble can be seen as a marker, which can be put down/picked up and moved by the automaton. Finally, Hoffman showed that the problem cannot be solved in neither finite nor co-finite labyrinths by using only one pebble [17]. Note that our goal differs from the one of labyrinth exploration, i.e., our goal is to visit all non-blocked cells.

3 Model

We consider the asynchronous version of the ANTS problem variant described in [12], where a set of mobile *agents* search the infinite grid for an adversarially hidden treasure. The agents are controlled by asynchronous finite state machines with a common sense of direction and communicate only with agents sharing the same grid cell.

More formally, we consider a set A of mobile agents that explore $\mathbb{Z}^2$. In the beginning of the execution, all agents are positioned in a designated grid cell referred to as the *origin*; the cell with coordinates $(0,0) \in \mathbb{Z}^2$. We denote the cells with either x- or y-coordinate being 0 as *north/east/south/west-axis*, depending on their location. The *distance* between two grid cells $(x,y),(x',y') \in \mathbb{Z}^2$ is defined with respect to the ℓ_1 norm (a.k.a. Manhattan distance), that is, $|x-x'|+|y-y'|$. Two cells are called *neighbors* if their distance is 1.

The set of cells $B \subset \mathbb{Z}^2$ represents the *blocked* cells, which cannot be entered by an agent. All other cells are called *free*. For simplicity, we assume that B neither contains the origin nor any of the cells within distance at most 3 from the origin. We note that assuming the origin free is necessary and that our protocols can easily be modified to work without assuming that the nearby cells around the origin are free. This assumption merely allows for a cleaner and more reader friendly initialization of our protocol.

To make the exploration of the grid feasible, we require that the cells in B do not fully enclose any free cell, i.e., any free cell is reachable from any other free cell by a path of free cells. The set B induces a set $\mathcal{O}$ of *obstacles*. An obstacle $O \in \mathcal{O}$ is a maximal set of connected cells, where two cells are connected if both their x- and y-coordinates each differ by at most one (diagonally adjacent cells are connected!). We require each obstacle to be of finite size.

All agents are controlled by the same asynchronous *finite automaton* (FA). This means that the individual agent has a constant memory and thus, in general, can not store coordinates in $\mathbb{Z}^2$. Since we design a protocol for a constant number of agents, we allow each agent to run a different individual protocol. This is modeled by assigning to each agent an individual initial state in the shared automaton. An agent a positioned in cell $z \in \mathbb{Z}^2$ can communicate with all other agents positioned in cell z at the same time. This communication is quite limited though: agent a merely senses for each state q of the finite state machine, whether there exists at least one agent $a' \neq a$ in cell z whose current state is q. In each step of the execution, agent a positioned in cell $(x,y) \in \mathbb{Z}^2$ can either move to one of the four neighboring cells $(x,y+1),(x,y-1),(x+1,y),(x-1,y)$, or stay put in cell (x,y). The former four *position* *transitions* are denoted by the corresponding cardinal directions $\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}$, whereas the latter (stationary) position transition is denoted by P. For convenience, we also identify the four directions $\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}$ with the unit vectors in the corresponding directions to be able to write, e.g., $z = (x, y) + \mathrm{N} = (x, y + 1)$. We point out that the agents have a common sense of orientation, i.e., the cardinal directions are aligned with the corresponding grid axes for every agent in every cell.

The agents operate in an asynchronous environment. Each agent's execution progresses in discrete (asynchronous) steps indexed by the non-negative integers and we denote the time at which agent a completed step $i > 0$ by $t_a(i) > 0$. Following the common practice, we assume that the time stamps $t_a(i)$ are determined by the policy ψ of an adversary that knows the protocol whereas the agents do not have any sense of time.

Formally, the agents' protocol is captured by the 3-tuple $\Pi = \langle Q, s_0^a, \delta \rangle$, where Q is the finite set of *states*; $s_0^a \in Q$ is the *initial state* of agent a; and

$$\delta : Q \times 2^Q \times \{\top, \bot\}^4 \to 2^{Q \times \{\mathrm{N,E,S,W,P}\}}$$

is the *transition function*. At time 0, all agents are positioned in the origin and their FAs are in the respective initial states. Suppose that at time $t_a(i)$, agent a is in state $q \in Q$ and positioned in cell $z \in \mathbb{Z}^2$. Then, the state $q' \in Q$ of a at time $t_a(i + 1)$ and its corresponding position transition $\tau \in \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}, \mathrm{P}\}$ are determined by the transition function $\delta(q, Q_a, b) = (q', \tau)$, where $Q_a \subseteq Q$ contains state $p \in Q$ if and only if there exists some (at least one) agent $a' \neq a$ such that a' is in state p and positioned in cell z at time $t_a(i)$, and b is a 4-tuple indicating which of the neighboring cells N/E/S/W are blocked ($\top$) or free ($\bot$). If the transition function dictates that an agent enters a blocked cell, the agent stays put instead. For simplicity, we assume that while the state subset Q_a (input to δ) is determined based on the status of cell z at time $t_a(i)$, the actual application of the transition function δ occurs instantaneously at the end of the step, i.e., agent a is considered to be in state q and positioned in cell z throughout the time interval $[t_a(i), t_a(i + 1))$.

The goal is to locate an adversarially hidden *treasure*, i.e., to bring at least one agent to the *free* cell in which the treasure is positioned. The distance to the treasure from the origin is denoted by D.

4 Basic Idea

In order to find the treasure, the agents have to visit every free cell. The high level idea is that the agents walk in growing squares counter-clockwise around the origin. To this end, each agent is given a specific task. An *explorer* explores the plane by walking along squares of increasing sizes, whereas four other agents, called *guides*, mark the four corners of the square that the explorer should walk along. We identify the four guides by the cardinal direction of their respective corner $\mathrm{NE}, \mathrm{NW}, \mathrm{SW}, \mathrm{SE}$. Upon entering a cell with a guide, the explorer accompanies the guide to the correct position for the next square before continuing the search. Please refer to Figure 1 for an illustration. After updating the position of the last guide, the explorer starts a new search along the next bigger square. We define *square*(d) as the square given by the four corner cells $(d, d), (d, -d), (-d, -d), (-d, d)$.

In the presence of obstacles, the subroutines get more involved. Obstacles can obstruct the path of the explorer or hinder a guide to mark the cell it is supposed to. To solve the former of the aforementioned problems we provide a subroutine that essentially allows the explorer to walk "through" the obstacle. For the second problem we change the conditions for the guides. Instead of marking the corner of the square, a guide has to either mark the correct

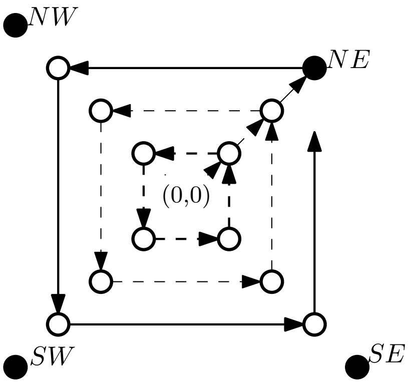


Figure 1 The filled black dots represent the corner agents (NW, SW, SE, NE), marking the next spot, where the explorer should turn counter-clockwise in order to walk a square. The hollow dots represent where the corner agents were in earlier stages. The arrows present the way the explorer was taking so far.

y-coordinate or the correct x-coordinate, depending on the guide. The NE- and SW-guides mark the y-coordinates of the corners of the square whereas the NW- and SE-guides mark the x-coordinates of said corners (see Figure 2).

Let us describe the new condition for the NE-guide. Consider the NE-guide that is supposed to mark the cell $z = (d, d)$ for some value of d and assume further that z is blocked. Then, the *surrogate cell* for the cell z is given by $z' = (x', d)$ where

$$x' = \min\{x \mid x \geq d \wedge (x, d) \notin B\} .$$

Informally, z' is the first free cell with the same y-coordinate as z further away from the origin. As the obstacles are of finite size we can guarantee that such a cell always exists. With this condition, we make sure that the guide is either on the corner (if it is free) or outside of the square on which the explorer is walking.

The condition for the other three guides is analogous. Consider square(d) and a guide responsible for the corner $M \in \{\mathrm{NE}, \mathrm{NW}, \mathrm{SW}, \mathrm{SE}\}$ of said square. Then, we denote by $Z_M(d)$ the cell where this guide will be positioned during the exploration of the square.

$$\begin{aligned}
Z_{\mathrm{NE}}(d) &= (x', d);\ x' = \min\{x'' \mid x'' \geq d \wedge (x'', d) \notin B\},\\
Z_{\mathrm{NW}}(d) &= (-d, y');\ y' = \min\{y'' \mid y'' \geq d \wedge (-d, y'') \notin B\},\\
Z_{\mathrm{SW}}(d) &= (x', -d);\ x' = \max\{x'' \mid x'' \leq -d \wedge (x'', -d) \notin B\},\\
Z_{\mathrm{SE}}(d) &= (d, y');\ y' = \max\{y'' \mid y'' \leq -d \wedge (d, y'') \notin B\}
\end{aligned}$$

5 Basic Capabilities

Our protocol requires the agents and in particular the explorer to be able to perform various advanced maneuvers. They have to be able to walk along the boundary of an obstacle, memorize their offsets from other cells, be able to find back to a cell they previously occupied, update the position of a guide to the next square, and, most importantly, to virtually walk through an obstacle. In this section we present the basic routines which are then combined in Section 6 to obtain the more complex ones.

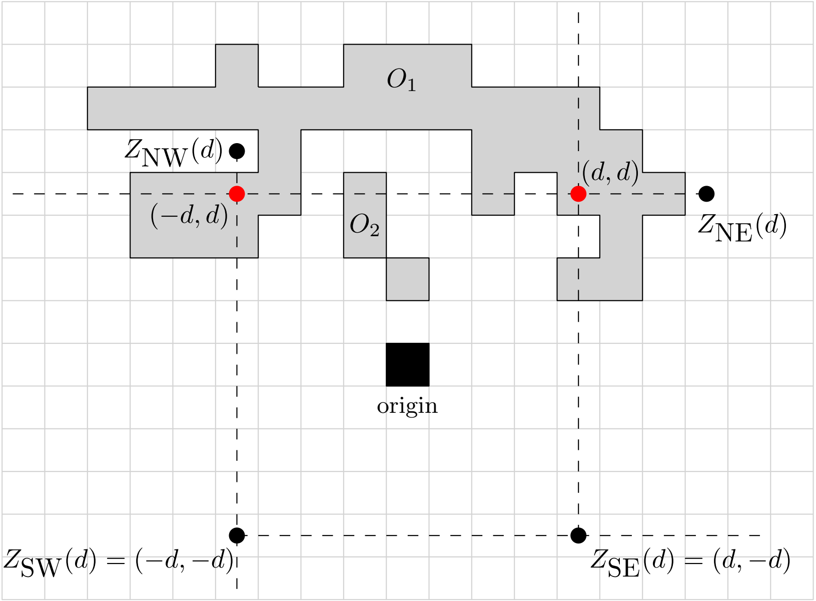


Figure 2 The grey area describes the obstacles O_1, O_2 and the red dots indicate where the NE- and NW-guide would be if there was no obstacle. The black dots indicate the cells, that the guides actually mark. The dashed lines indicate the side of the square that the respective guide is marking and altogether mark the square that the explorer is supposed to walk along.

5.1 Walking Around an Obstacle

Consider an agent a that currently walks into direction h where h can be N/E/S/W and is called the *heading* of a. We say that a turns right or left as shorthand for a changing its heading to an adjacent cardinal direction. Now suppose that agent a is in cell $z = (x, y)$ and the cell $z + h$ is blocked by the obstacle O that a intends to walk around. In the very first step, a turns right so that the obstacle is on its left side – an invariant that will be maintained during the process of walking around the obstacle. Then, in every following step, a first checks if the cell on the left side with respect to the current heading is blocked. If this is the case, a walks once towards its heading, if possible. In case the cell towards the heading is also blocked, a turns right. In the case that the cell on the left is free, a turns left and walks once towards the new heading. We can verify that this case only occurs if in the previous step, a moved towards its current heading and therefore, the cell on the left was blocked. Therefore, the obstacle will again be on the left side of a in the next step. The details of the method StepCounterClockwise for a single step are given in Procedure 1. The method assumes that agent a is positioned in a cell along the border of the obstacle O and the cell left of a (with respect to h) is blocked by the obstacle O. As the procedure ensures the aforementioned invariant, agent a can execute it repeatedly to traverse the complete boundary of the obstacle.

5.2 Bounded Offset Counter

In this section we explain how the agents can simulate a bounded counter. As the agents have only a constant number of states, they can not remember arbitrarily large numbers, such as how many steps north they went along an obstacle. In order to circumvent this lack of memory, the ants collaboratively implement one or more offset counters. The counter is suitable to memorize offsets to cells while moving along the boundary of an obstacle. The counter provides the basic operations On, Off, IsNull, IsPositive, IsNegative, Increment, and Decrement, which activate/deactivate the counter, allow the agent to determine whether the offset is zero/positive/negative, or to increment/decrement it, respectively. It is important to note that our implementation of the offset counter is only available while the agent is adjacent to an obstacle and while this obstacle stays the same. As soon as the agent moves to a cell that is not adjacent to the obstacle anymore, the value of the counter becomes invalid. Hence, our protocols ensure that the counter is always turned off before leaving an obstacle. Moreover, the value of the counter only works correctly as long as its value is bounded by the circumference of the obstacle. This does not pose a problem, however, as all offsets that the agents need to store in our protocol are bounded appropriately.

Procedure 1: StepCounterClockwise()

```
Agent $a$ is located in $(x, y)$ and has heading $h$
if cell on left is free then
    turn left
else if $(x, y) + h$ is blocked then
    while $(x, y) + h$ is blocked do
        turn right
move once towards $h$
return $h$
```

We first give an informal description of our implementation and then specify how the basic operations can be implemented. Consider an agent a located in a cell (x, y) adjacent to an obstacle O. Agent a is equipped with the counter c represented by the auxiliary agents a_c, a_b, and a_m called *count agent*, *base agent*, and *messenger agent*, respectively. When the counter is turned off, the auxiliary agents are in the *follow mode*, which implies that they simply follow agent a and do not perform any specific task. When the counter is turned on, the auxiliary agents enter the *counter mode* and perform special tasks. The job of a_b is to mark the cell where the counter has been turned on the last time. Agent a_c's task is to store an offset value v by residing in the cell that is reached when starting in the cell containing a_b and walking $|v|$ cells clockwise along the boundary of the obstacle O. In order to distinguish positive and negative offsets, a_c encodes the sign of v in its states. Agent a_m generally resides in the same cell as agent a and moves to a_c and a_b when the counter is to be changed or read. Either of the basic operations can only be executed when the previous operation has been completed, which is the case when a_m is in the same cell as a.

For the purpose of argumentation, we denote the value represented by counter c as $\mathit{val}(c)$. We remark, however, that this value is not directly accessible to any of the agents.

Operation On(c). When a activates the counter, it signals this to the auxiliary agents using a special state, upon which they enter their respective counter mode states.

Operation Off(c). Agent a_m moves clockwise around the obstacle, when it meets a_c and a_b it instructs them to move along the obstacle to the cell containing a, and finally does the same. The auxiliary agents then enter the follow mode.

Operation IsNull(c). Agent a_m walks clockwise until it locates the cell containing agent a_b. It checks whether agent a_c occupies the same cell and reports this information to agent a.

Operation IsPositive/IsNegative(c). Agent a_m walks clockwise until it locates the cell containing the agent a_c. If the cell also contains agent a_b – the value of the counter is zero – agent a_m reports `false` to a . Otherwise, a_m senses the sign of c through the state of a_c and reports the result to a accordingly.

Operation Increment/Decrement(c). Agent a_m walks clockwise until it locates the cell containing agent a_c. It then instructs a_c to increment/decrement and returns to agent a. Depending on whether the state of a_c corresponds to a positive or negative sign, a_c moves one cell clockwise or counter-clockwise along the obstacle. If a_c resides in the same cell as a_b, it also needs to change its sign state accordingly.

These operations complete the specification of the counter functionality. Please note that all these operations make only use of a constant number of states.

5.3 Combining Offset Counters

The agents in our protocol sometimes employ a constant number of offset counters c_1 to c_k on the same obstacle, where the respective counters are activated in the same cell. This functionality can be provided by having one base agent a_b and one messenger agent a_m and k count agents for the different counters. To ensure that the messenger interacts with the correct count agent, they encode an index in their states such that the messenger agent can distinguish them. Correspondingly, the messenger agent encodes the index of the counter that it is operating on in its state. As only a constant number of offsets are used, this is possible with a constant finite automaton. We distinguish the count agents of different counters by their index as superscript, i.e., a_c^i is the count agent of the counter c_i.

When an agent uses several counters, it has access to two additional operations. Operation $\textsc{LessThan}(c_i, c_j)$ compares the value of two counters and returns a boolean indicating whether $\text{val}(c_i) < \text{val}(c_j)$. The operation $\textsc{Set}(c_i, c_j)$ sets the value of counter c_i to $\text{val}(c_j)$.

Operation LessThan(c_i, c_j). Agent a_m moves clockwise around the obstacle until it locates the cell containing a_b. Then, a_m walks further clockwise around the obstacle until having located both a_c^i and a_c^j. Based on the signs encoded in the states of a_c^i and a_c^j and the order in which these agents were located, a_m infers the result of the comparison, then returns to a and signals it.

Operation Set(c_i, c_j). Agent a_m walks along the obstacle to the cell containing a_c^i and instructs a_c^i to walk to the cell containing a_c^j, while a_m accompanies a_c^i on its way. When a_c^i enters the cell containing a_c^j, agent a_c^i updates its sign to the sign of a_c^j and agent a_m returns to a to finish the operation.

6 Advanced Procedures

In this section, we combine the basic functionalities described in the previous section into the complex procedures, that eventually constitute our search protocol. The most important functionality is the ability to virtually walk through an obstacle following a horizontal or vertical straight line. The agents do this by locating the closest cell that lies on the straight

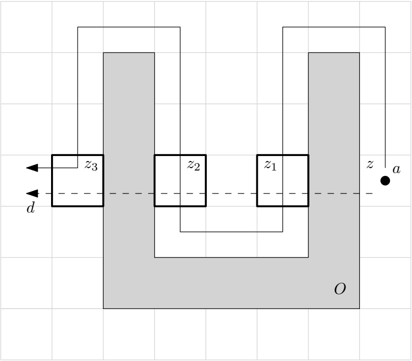


Figure 3 Agent a wants to walk west but the direct path (dashed arrow) is obstructed by an obstacle O. Thus, a walks counter-clockwise around the boundary of O (continuous arrow) and uses offset counters to detect the potential target cells z_1, z_2, and z_3.

line through the obstacle and then continue the walk from there. This functionality is realized by the procedures Shift and Probe that will be described next.

6.1 Shifting the Position Along an Obstacle

The procedure Shift(c_x, c_y) allows an agent a positioned in cell $z = (x, y)$ next to the obstacle O and equipped with two counters c_x and c_y to move to the cell $z' = (x + \text{val}(c_x), y + \text{val}(c_y))$, where z' must be also next to O. During the process, agent a continuously updates the counters to reflect the new offsets, so that when a has reached cell z', the values of both counters c_x and c_y are zero. Consequently, both counters are then turned off. Procedure 2 gives a pseudo-code description.

```
Procedure 2: Shift($c_x, c_y$)
  while $\neg$IsNull($c_x$) $\vee$ $\neg$IsNull($c_y$) do
      $h \leftarrow$ StepCounterClockwise()
      Increment($c_x$) / Decrement($c_x$) according to $h$
      Increment($c_y$) / Decrement($c_y$) according to $h$
  Off($c_x$); Off($c_y$)
```

6.2 Probing Target Cells

While the procedure StepCounterClockwise allows the agent a to walk around an obstacle O, it still needs to figure out which of the cells visited along the walk is the next free cell t along the straight path through O. There are two main difficulties that we face when trying to identify t. First, the circumference of O can be arbitrarily large and therefore, a single agent cannot keep track of its relative location with respect to its starting cell $z = (x_b, y_b)$. Second, there might be many possible cells along the edges of O that are hit by the straight line through O. We refer to all these cells along the border of O as *potential target cells* (cf. Figure 3).

The procedure Probe allows an agent a in cell z to locate the closest potential target cell z^* in direction of its heading h and returns a counter representing the distance of z^* relative to z. The exact formulation of Probe depends on the heading h of a. Procedure 3 gives a pseudo-code description for the case of $h = W$. The other cases are analogous.

The idea is that agent a employs three counters c_x, c_y and $c_{\min}$ while walking along the boundary of O. The counters c_x and c_y track the offset of a from the initial cell (x_b, y_b). Whenever c_y is zero, a has located a cell with the same y-coordinate and the value of c_x is stored in $c_{\min}$ if it is smaller than the previous $c_{\min}$. This process is iterated until the agent returns to the starting position (it meets agent a_b again). Then it turns off counters c_x and c_y and returns $c_{\min}$.

```
Procedure 3: Probe_W()
  On($c_x$); On($c_y$); On($c_{\min}$);
  repeat
      $h \leftarrow$ StepCounterClockwise()
      Increment($c_x$) / Decrement($c_x$) according to $h$
      Increment($c_y$) / Decrement($c_y$) according to $h$
      if IsNull($c_y$) $\wedge$ (IsNull($c_{\min}$) $\vee$ LessThan($c_x, c_{\min}$)) then
          Set($c_{\min}, c_x$)
  until $a$ meets $a_b$;
  Off($c_x$); Off($c_y$);
  return $c_{\min}$
```

6.3 Procedure Scan

A detail that we have to be careful with is, when traveling from one guide to another, that each cell along the current square gets discovered and that we eventually reach the guide. To this end, the explorer visits each cell on the boundary of an obstacle that it meets using the procedure Scan.

Upon executing Scan, agent a first activates two counters c_x and c_y. Then, it walks once around the obstacle by repeatedly invoking StepCounterClockwise and updating c_x and c_y according to its movements. If a meets the next guide along the way, it does not update the counters anymore. When a returns to the cell containing the base agent a_b of its counter, the walk is finished. If both c_x and c_y equal 0, no guide was not found during Scan. Otherwise, the values of the counters represent the offset to the guide and the procedure "returns" the two counters c_x and c_y. Since a might meet different guides, it stores the index of the next guide that it is supposed to meet according to the protocol in its state, thereby allowing it to ignore all other guides.

6.4 Procedure Update

In this section, we establish the procedure Update that allows the explorer to find the cell $Z_M(d+1)$ starting from the cell $Z_M(d)$ of some guide M for any $d > 0$. Our goal is to prove the following lemma.

▶ **Lemma 1.** *The procedure* Update(M) *enables the the explorer to move from cell $Z_M(d)$ to cell $Z_M(d+1)$ and back to cell $Z_M(d)$, for any $d \geq 1$.*

Consider Update in the case of the NW-guide currently occupying cell $Z_{\mathrm{NW}}(d) = (-d, y^*)$. We assume that the explorer has access to a counter c_y, denoting the y-offset to the line $y = d$. To initialize the update, the explorer leaves another agent to mark $Z_{\mathrm{NW}}(d)$ and instructs the NW-guide to follow the explorer. A lengthy pseudo-code representation of the Update for the NW-guide can be found in Procedure 4, where Update$_{\mathrm{NW}}(c)$ stands for the special case of the NW-guide. To locate the cell $Z_{\mathrm{NW}}(d+1)$, our first task is to find a cell $z \in L$, where L is the set of cells whose x-coordinate equals to $-(d+1)$ and whose y-coordinate is at least as large as $d+1$, i.e.,

$$L = \{(i,j) \in \mathbb{Z}^2 \mid (i = -(d+1)) \wedge (j \geq d+1))\} .$$

We divide our description of Update into several cases. First, we consider the case that the cell $z_w = (-(d+1), y^*)$ west to $Z_{\mathrm{NW}}(d)$ is blocked by obstacle O. This induces that $Z_{\mathrm{NW}}(d)$ and $Z_{\mathrm{NW}}(d+1)$ are on the border of the same obstacle O. Refer to Figure 4b for an illustration. The explorer turns on the c_x counter. Then, it increments its value by 1 to correspond to the offset from $Z_{\mathrm{NW}}(d+1)$. Also the c_y counter is decremented by 1, to mark the next desired y-coordinate.

To reach a cell $z \in L$, the explorer now simply turns its heading to north to initialize a walk counter-clockwise around O. Now since O is finite, it has to be the case that there is at least one cell from L on the boundary of O. The explorer successively executes StepCounterClockwise, updates counters c_x and c_y accordingly, and always checks if $c_x = 0$ and if c_y is positive. If the check returns true, the explorer has reached a cell $z \in L$.

To now find the cell $Z_{\mathrm{NW}}(d+1)$, the explorer first turns its heading towards south and then successively executes Shift(0, Probe()), and updates c_y accordingly during every Shift, until Probe returns a value greater than the current c_y. If the next cell found by Probe is further away than c_y we know that we are in the cell $Z_{\mathrm{NW}}(d+1)$ at the moment. As the last step of this case, the explorer instructs the NW-guide to remain in this cell, and walks counter-clockwise around O until it finds the agent denoting cell $Z_{\mathrm{NW}}(d)$.

Then, consider the case that cell z_w is not blocked. We further split into two cases and we first consider the case that $c_y > 0$, which can be asserted by the explorer by checking if IsPositive(c_y) returns true. Then, it has to be the case that all cells $(-d, y^* - i)$, for $i \leq y^* - d$, are blocked by some obstacle O due to the invariant that $Z_{\mathrm{NW}}(d)$ has the smallest y-coordinate among free cells $(-d, y \geq d)$. See Figure 4a for an illustration. Thus, the explorer can move to z_w without invalidating the counter c_y. Furthermore, cell $Z_{\mathrm{NW}}(d+1)$ has to be on the boundary of O.

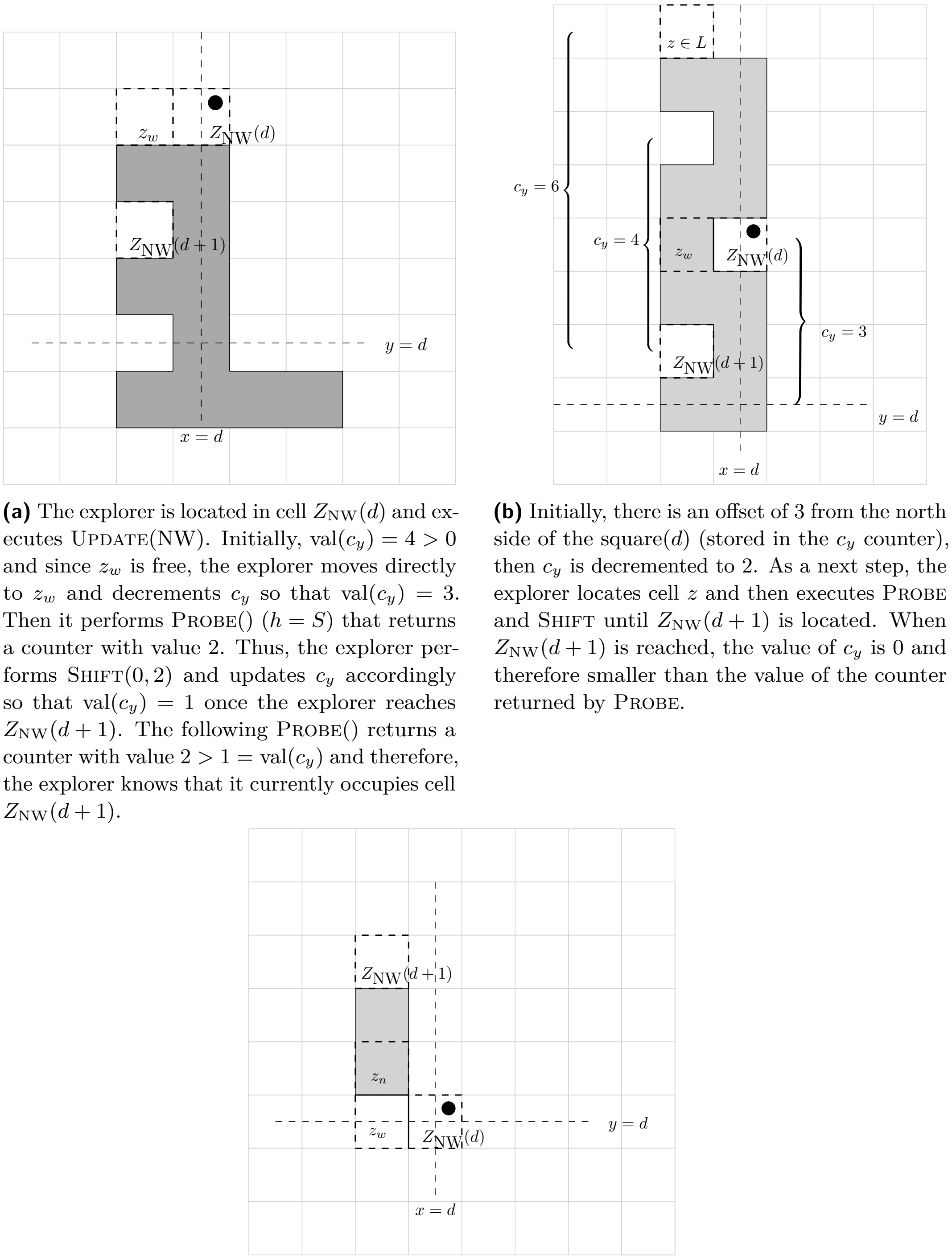
Next, the explorer decrements c_y by 1. If $c_y = 0$, then we have reached cell $Z_{\mathrm{NW}}(d+1)$. Otherwise, similarly to the previous case, the explorer now turns its heading towards south and executes Shift(0, Probe()) until Probe returns a value greater than c_y. When Probe returns a value greater than c_y, the explorer has reached cell $Z_{\mathrm{NW}}(d+1)$. Similarly to the previous case, the explorer instructs the NW-guide to mark this cell and travels back to $Z_{\mathrm{NW}}(d)$ by walking around obstacle O.

Consider now the case where z_w is not blocked and $c_y \leq 0$. Note that due to the invariant that $Z_{\mathrm{NW}}(d)$ has the smallest y-coordinate among free cells $(-d, y \geq d)$, we get that $c_y = 0$. Therefore, the explorer can turn off both counters c_x and c_y without losing any information. Then, the explorer along with the other agents, moves to cell z_w. After reaching z_w, the explorer turns its heading towards north and if $(-(d+1), d+1)$ is not blocked, it moves once north reaching the cell $Z_{\mathrm{NW}}(d+1)$. After instructing NW to mark $Z_{\mathrm{NW}}(d+1)$, the explorer can find back to $Z_{\mathrm{NW}}(d)$ simply by reversing its movements.

If $(-(d+1), d+1)$ is blocked, then the explorer executes Shift(0, Probe()) once, so that it reaches the free cell with the smallest y-coordinate at least $d+1$, i.e., the cell $Z_{\mathrm{NW}}(d+1)$. Refer to Figure 4c in the appendix for an illustration. The explorer again instructs the NW-guide to remain in $Z_{\mathrm{NW}}(d+1)$ and travels back to $Z_{\mathrm{NW}}(d)$ by turning its heading south, executing Shift(0, Probe()) once, and moving once east.

In all of the above cases, the guide was left in a cell $Z_{\mathrm{NW}}(d+1)$ yielding the correctness of the update procedure for the NW-guide and the explorer found its way back to the cell

```
Procedure 4: UpdateNW($c_y$)
  Agent $a$ is located in $Z_{\mathrm{NW}}(d) = (-d, y^*)$, $z_w = (-(d+1), y^*)$, $z_n = (-(d+1), d+1)$
  Mark $Z_{\mathrm{NW}}(d)$ with an agent $a_{mark}$
  if $z_w \in O$ then
      ▷ $z_w \in O \Rightarrow$ $Z_{\mathrm{NW}}$(d) and $Z_{\mathrm{NW}}$(d+1) are next to the same obstacle
      ▷ Figure 4b represents this case
    $h \leftarrow$ N; On($c_x$); Increment($c_x$); Decrement($c_y$);
      ▷ store offsets to the coordinate (-(d+1),d+1) instead to (-d,d)
    repeat
      $h \leftarrow$ StepCounterClockwise;
      Increment($c_x$) / Decrement($c_x$) according to $h$;
      Increment($c_y$) / Decrement($c_y$) according to $h$:
    until IsNull($c_x$) $\wedge$ IsPositive($c_y$);
      ▷ We found the cell z, cell $Z_{\mathrm{NW}}$(d+1) is south to us
    $h \leftarrow$ S; Off($c_x$); On($c_0$);
    repeat
      $c_{y'} \leftarrow$ Probe();
      Shift($c_0, c_{y'}$) while updating $c_y$;
    until LessThan($c_y, c_{y'}$);
    turn off all counters; leave the NW-guide in this cell; follow the obstacle back to
     $c_{mark}$;
  else
    $h \leftarrow$ W; move once towards $h$;  ▷ $z_w$ is free, walk one step west
    if IsPositive($c_y$) then
        ▷ (-d/d) is blocked and $Z_{\mathrm{NW}}$(d) is further north
        ▷ $Z_{\mathrm{NW}}$(d) and $Z_{\mathrm{NW}}$(d+1) are next to the same obstacle
      Decrement$c_y$;
      if $\neg$IsNull($c_y$) then
          ▷ We are further north than needed for $Z_{\mathrm{NW}}(d+1)$
          ▷ Figure 4a represents this case
        $h \leftarrow$ S; Off($c_x$); On($c_0$), $c_{y'} \leftarrow$ Probe()
        while LessThan($c_y, c_{y'}$) do
          $c_{y'} \leftarrow$ Probe()
          Shift($c_0, c_{y'}$) while updating $c_y$
      turn off all counters; leave the NW-guide in this cell, follow the obstacle back to
       $c_{mark}$
    else
      Off($c_x$); Off($c_y$); On($c_y$); Decrement($c_y$);
      $h \leftarrow$ N
      if $z_n \in O$ then
          ▷ $(-d, d)$ is free, $z_n$ is blocked, see Figure 4c
        On($c_0$); $c_{y'} \leftarrow$ Probe()
        Shift($c_0, c_{y'}$)
        turn off all counters; leave the NW-guide in this cell; reverse the movements
         to go back to $c_{mark}$
      else
          ▷ $(-d, d)$ and $(-(d+1), d+1)$ are both free
        move once towards $h$, leave the NW-guide in this cell
        turn off all the counters; move once south and once east to go back to $c_{mark}$
```



(a) The explorer is located in cell $Z_{\mathrm{NW}}(d)$ and executes UPDATE(NW). Initially, $\mathrm{val}(c_y) = 4 > 0$ and since z_w is free, the explorer moves directly to z_w and decrements c_y so that $\mathrm{val}(c_y) = 3$. Then it performs PROBE() ($h = S$) that returns a counter with value 2. Thus, the explorer performs SHIFT(0, 2) and updates c_y accordingly so that $\mathrm{val}(c_y) = 1$ once the explorer reaches $Z_{\mathrm{NW}}(d + 1)$. The following PROBE() returns a counter with value $2 > 1 = \mathrm{val}(c_y)$ and therefore, the explorer knows that it currently occupies cell $Z_{\mathrm{NW}}(d + 1)$.

(b) Initially, there is an offset of 3 from the north side of the square(d) (stored in the c_y counter), then c_y is decremented to 2. As a next step, the explorer locates cell z and then executes PROBE and SHIFT until $Z_{\mathrm{NW}}(d + 1)$ is located. When $Z_{\mathrm{NW}}(d + 1)$ is reached, the value of c_y is 0 and therefore smaller than the value of the counter returned by PROBE.

(c) The first cell to the west from $Z_{\mathrm{NW}}(d)$ is free, c_y equals 0, and $Z_{\mathrm{NW}}(d + 1)$ is located by moving once west and then executing PROBE and SHIFT with heading N.

Figure 4 Special cases of UPDATE.

$Z_{\mathrm{NW}}(d)$. This concludes the description of Update for the NW-guide. The procedure Update works analogously for other guides. Note that when updating the NE-guide, the explorer does not return back to cell $Z_{\mathrm{NE}}(d)$ and therefore does not leave an agent in that cell. Thus, Lemma 1 follows.

7 Searching the Plane

In the search protocol SquareWalk, the agents begin the search by four agents moving into the cells $(1,1)$, $(-1,1)$, $(-1,-1)$, and $(1,-1)$, corresponding to $Z_{\mathrm{NE}}(1)$, $Z_{\mathrm{NW}}(1)$, $Z_{\mathrm{SW}}(1)$, and $Z_{\mathrm{SE}}(1)$. Recall that these agents, the guides, essentially mark the corners of the square that the explorer will explore next and that we identify each guide with the cardinal direction of its corner (NE, NW, SW, SE). The explorer e, equipped with a set of counters in follow mode, moves to the NE-guide in the cell $Z_{\mathrm{NE}}(1)$. It then starts to explore square(1) by moving west until it meets the NW-guide in cell $Z_{\mathrm{NW}}(1)$ and, together with the NW-guide, moves to cell $Z_{\mathrm{NW}}(2)$. Then the explorer returns to $Z_{\mathrm{NW}}(1)$ and moves south towards the SW-guide. It proceeds analogously with the other guides and eventually returns to the NE-guide. After moving the NE-guide to cell $Z_{\mathrm{NE}}(2)$, the explorer does *not* return to $Z_{\mathrm{NE}}(1)$ but instead starts to explore square(2). Starting from the next iterations, things get more involved as obstacles might obstruct the explorer or the guides. Consider the situation that the next square to be searched by the explorer is square(d), every guide M is in the corresponding cell $Z_M(d)$, and the explorer is in cell $Z_{\mathrm{NE}}(d)$. We explain how e can walk from the NE-guide to the NW-guide while exploring the north side of square(d); the three other sides of the square are analogous. Procedure 5 gives a pseudo-code description in which $z_e = (x_e, y_e)$ denotes the current cell of the explorer while an explanation follows below.

```
Procedure 5: ExploreNorthSide
  $h \leftarrow \mathrm{W}$    ▷ set heading
  repeat
      if $(z_e + h) \notin B$ then
          move($h$)    ▷ next cell is free
      else
          $c_{\mathrm{probe}} \leftarrow$ Probe()
          $(c_x, c_y) \leftarrow$ Scan()
          if (IsNull($c_x$) ∧ IsNull($c_y$)) ∨ LessThan($c_{probe}, c_x$) then
              Off($c_y$); On($c_y$); Off($c_x$)    ▷ reset $c_y$ to zero and turn off $c_x$
              Shift($c_{\mathrm{probe}}, c_y$)    ▷ move to next free cell
          else
              Off($c_{\mathrm{probe}}$); On($c_{\mathrm{update}}$)    ▷ re-use agents from the $c_{\mathrm{probe}}$ counter
              Set($c_{\mathrm{update}}, c_y$)
              Shift($c_x, c_y$)    ▷ move to NW-guide
  until $e$ meets NW;
  Update(NW, $c_{\mathrm{update}}$)
```

The explorer e sets its heading towards west and, as long as the cell in front is free, moves forward. If e senses an obstacle in front in cell z, e executes Probe to find the next free cell z' in the direction of its heading, resulting in the counter c_{probe} representing the distance between z_e and z'. Then e scans the obstacle using Scan yielding the counters c_x and c_y. If

Scan was not successful, i.e., the NW-guide was not located along the obstacle, the counters c_x and c_y are both zero. Now, e moves to z' using $\textsc{Shift}(c_{\text{probe}}, 0)$ (c_y is reset and used as second parameter) if

(i) Scan was not successful, i.e., the NW-guide was not located along the obstacle (corresponding to the case that $(\textsc{IsNull}(c_x) \wedge \textsc{IsNull}(c_y) = \texttt{true})$ or

(ii) Scan found the next guide but it is further west than the next target cell (corresponding to the case that $\textsc{LessThan}(c_{\text{probe}}, c_x) = \texttt{true}$)

and repeats the above. If $\text{val}(c_{\text{probe}}) \geq \text{val}(c_x)$, corresponding to $\textsc{LessThan}(c_{\text{probe}}, c_x) = \texttt{false}$, the explorer executes $\textsc{Shift}(c_x, c_y)$ to move to Z_{NW} to meet the NW-guide.

Finally, e uses Update to update the position of the NW-guide from $Z_{\text{NW}}(d)$ to $Z_{\text{NW}}(d{+}1)$ and returns to $Z_{\text{NW}}(d)$. Then, it sets its heading to south, turns off all counters and starts the analogous procedure ExploreWestSide, this time walking south towards the SW-guide.

The above procedure is repeated for all four sides of the square until the explorer arrives back at the NE-guide and updates its position to $Z_{\text{NE}}(d+1)$. Now e does *not* return to $Z_{\text{NE}}(d)$ but instead starts a search of square$(d+1)$ using ExploreNorthSide.

7.1 Correctness

In this section, we establish the correctness of the protocol SquareWalk, i.e., that it guarantees that the explorer eventually visits all free cells of the grid. We define the concept of a *configuration* $C : A \mapsto \mathbb{Z}^2$ as an assignment of a cell to each agent. A configuration is a snapshot of the positions of the agents at a given time. The *start configuration for distance* d, denoted by $\mathbf{Z}(d)$, is the configuration where each guide M is in its corresponding cell $Z_M(d)$ and the explorer and the auxiliary agents are in cell $Z_{\text{NE}}(d)$ with the NE-guide. We furthermore define

$$F_i = \{(x, y) \notin B \mid (|x| = i \wedge |y| \leq i) \vee (|y| = i \wedge |x| \leq i)\}$$

as the set free cells of square(i) for some $i \geq 1$. We are now ready to prove the following theorem which establishes the correctness of SquareWalk. Due to the space constraints, we defer the proof to the full version of the paper.

▶ **Theorem 2.** *The protocol* SquareWalk *guarantees that every free cell $z \in \mathbb{Z}^2$ is visited by the explorer within finite time.*

8 Conclusion

We presented the protocol SquareWalk that allows a group of finite state machines (with a constant number of states) to locate an adversarially hidden treasure in a plane obstructed by arbitrary obstacles of finite circumference. Our search protocol employs the weak communication capabilities of the agents to simulate a sufficient amount of memory to ensure progress in the search.

Our search protocol requires ten agents in total, where one of the agents acts as an explorer, who performs the searching. The protocol uses three offset counters, requiring five agents. The other four agents mark the sides of a square around the origin that bounds the area discovered so far. We note that we can reduce the agent count to nine by using the triangle approach from [11]. But as this makes the specification of our protocol considerably more involved, we presented the simpler version employing the square approach.

References

1 Susanne Albers and Monika Henzinger. Exploring Unknown Environments. *SIAM Journal on Computing*, 29:1164–1188, 2000.

2 Baruch Awerbuch and Margrit Betke. Piecemeal Graph Exploration by a Mobile Robot. *Information and Computation*, 1999.

3 Michael Bender, Antonio Fernandez, Dana Ron, Amit Sahai, and Salil Vadhan. The Power of a Pebble: Exploring and Mapping Directed Graphs. In *Proceedings of the 30th annual ACM Symposium on Theory of Computing (STOC)*, 1998.

4 Manuel Blum and Dexter Kozen. On the Power of the Compass (or, Why Mazes Are Easier to Search Than Graphs). In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 132–142, 1978.

5 Manuel Blum and William J. Sakoda. On the Capability of Finite Automata in 2 and 3 Dimensional Space. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 147–161, 1977.

6 Lothar Budach. Automata and Labyrinths. *Mathematische Nachrichten*, pages 195–282, 1978.

7 Xiaotie Deng and Christos Papadimitriou. Exploring an Unknown Graph. *Journal of Graph Theory*, 32:265–297, 1999.

8 Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. Tree Exploration with Little Memory. *Journal of Algorithms*, 51:38–63, 2004.

9 Klemens Döpp. Automaten in Labyrinthen. *Elektronische Informationsverarbeitung und Kybernetik*, 7(2):79–94, 1971.

10 Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal Constrained Graph Exploration. *ACM Transactions on Algorithms (TALG)*, 2(3):380–402, 2006. doi:10.1145/1159892.1159897.

11 Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How Many Ants Does it Take to Find the Food? In *21th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 263–278, 2014.

12 Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS Problem with Asynchronous Finite State Machines. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 471–482, 2014.

13 Ofer Feinerman and Amos Korman. Memory Lower Bounds for Randomized Collaborative Search and Implications for Biology. In *Proceedings of the 26th International Conference on Distributed Computing (DISC)*, pages 61–75, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-33651-5_5.

14 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative Search on the Plane Without Communication. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 77–86, 2012.

15 Pierre Fraigniaud and David Ilcinkas. Digraphs Exploration with Little Memory. In *21st Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 246–257, 2004.

16 Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph Exploration by a Finite Automaton. *Theoretical Computer Science*, 345(2-3):331–344, 2005.

17 Frank Hoffmann. One Pebble Does Not Suffice to Search Plane Labyrinths. In *Fundamentals of Computation Theory*, pages 433–444. Springer Berlin Heidelberg, 1981.

18 Christoph Lenzen, Nancy Lynch, Calvin Newport, and Tsvetomira Radeva. Trade-offs between Selection Complexity and Performance when Searching the Plane without Communication. In *Proceedings of the 33rd Symposium on Principles of Distributed Computing (PODC)*, pages 252–261, 2014.

19 Saket Navlakha and Ziv Bar-Joseph. Distributed Information Processing in Biological and Computational Systems. *Communications of the ACM*, 58(1):94–102, 2014.

20 Petrişor Panaite and Andrzej Pelc. Exploring Unknown Undirected Graphs. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 316–322, 1998.

21 Noa Pinter-Wollman, Ashwin Bala, Andrew Merrell, Jovel Queirolo, Martin C Stumpe, Susan Holmes, and Deborah M Gordon. Harvester Ants Use Interactions to Regulate Forager Activation and Availability. *Animal behaviour*, 86(1):197–207, 2013.

Distributed Sparse Cut Approximation*

Fabian Kuhn[1] **and Anisur Rahaman Molla**[2]

1 **Department of Computer Science, University of Freiburg, Freiburg, Germany**
kuhn@cs.uni-freiburg.de

2 **Department of Computer Science, University of Freiburg, Freiburg, Germany**
armolla@cs.uni-freiburg.de

Abstract

We study the problem of computing a sparse cut in an undirected network graph $G = (V, E)$. We measure the sparsity of a cut $(S, V \setminus S)$ by its conductance $\phi(S)$, i.e., by the ratio of the number of edges crossing the cut and the sum of the degrees on the smaller of the two sides. We present an efficient distributed algorithm to compute a cut of low conductance. Specifically, given two parameters b and ϕ, if there exists a cut of balance at least b and conductance at most ϕ, our algorithm outputs a cut of balance at least $b/2$ and conductance at most $\tilde{O}(\sqrt{\phi})$, where $\tilde{O}(\cdot)$ hides polylogarithmic factors in the number of nodes n. Our distributed algorithm works in the CONGEST model, i.e., it only requires to send messages of size at most $O(\log(n))$ bits. The time complexity of the algorithm is $\tilde{O}(D + 1/b\phi)$, where D is the diameter of G. This is a significant improvement over a result by Das Sarma et al. [ICDCN 2015], where it is shown that a cut of the same quality can be computed in time $\tilde{O}(n + 1/b\phi)$. The improved running time is in particular achieved by devising and applying an efficient distributed algorithm for the all-prefix-sums problem in a distributed search tree. This algorithm, which is based on the classic parallel all-prefix-sums algorithm, might be of independent interest.

1998 ACM Subject Classification F.2.2 [Analysis of Algorithms and Problem Complexity] Non-numerical Algorithms and Problems, G.2.2 [Discrete Mathematics] Graph Theory

Keywords and phrases sparsest cut, conductance, random walks, all-prefix-sums

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.10

1 Introduction and Related Work

The problem of finding sparse cuts in a graph is one of the basic problems in network optimization. In the context of the present paper, the sparsity of a cut is measured by its *conductance*, where the conductance of a cut is defined as the ratio between the number of edges crossing the cut and the sum of the degrees on the smaller side of the cut. In this context, the sum of the degrees of a set of nodes S is also known as the *volume* of S. The conductance of a graph is defined as the smallest conductance of any of its cuts. The conductance determines how well connected a graph is and in particular how well information can be spread within the graph. It is well-known that the conductance of a graph is closely connected to the *mixing time* of a random walk on the graph and consequently also to the *spectral gap* of the graph [14, 20]. Network with high conductance, a large spectral gap and thus a small random walk mixing time for example allows to do fast (almost) uniform random sampling of nodes (see [11] for fast distributed algorithms and applications) or to do low-congestion routing [7, 13]. It is also known that the performance of the random push-pull gossip protocol is very closely related to the conductance of the network [12].

* Research supported by ERC Grant No. 336495 (ACDC).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 10; pp. 10:1–10:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As the conductance of a graph is tightly connected to the performance of many important random processes and computations in networks, finding cuts of low conductance potentially helps to find lower bounds on the performances of these processes and computations. As it is the sparse cuts which are limiting the speed of many such processes, finding low conductance cuts can help in identifying critical, important edges and bottlenecks in a given network. In a standard centralized setting, the problem of finding sparse cuts and also more generally related graph partitioning problems are well studied with a large body of literature, see e.g., [1, 2, 3, 4, 5, 15, 18, 19, 21, 24]. The first approximation algorithm the conductance of a graph was presented by Leighton and Rao in [18], where an $O(\log n)$-approximation is given. Using semidefinite programming techniques, this was improved to the currently best known approximation ratio of $O(\sqrt{\log n})$ Arora, Rao, and Vazirani [4] (see also [3]).

In a large-scale network, it might not be possible or reasonable to collect the entire structure of the network at a single node and to perform computations in a centralized way. In the present paper, we thus study the problem of finding cuts of low conductance in a distributed manner. Our distributed algorithm is based on random walk techniques that were first developed by Lovász and Simonovits in [19, 20] and later extended by Spielman and Teng in [24, 25]. More directly, our distributed algorithm is based on an algorithm for the same problem in the streaming model [9] and on a simple distributed version of this algorithm which appeared in [10].

The core idea of the algorithm is to test different cuts obtained by the probability distributions of random walks in the graph. More specifically, consider a random walk of some length ℓ starting at a node s in a graph G and order all nodes of G according to a normalized probability of finishing the random walk at the node. Consider all $n-1$ cuts that are defined by all the $n-1$ prefixes of this ordering. Assume that there is a set of nodes S of volume at most half the total volume of G such that the $(S, \bar{S})$ of G has conductance at least ϕ. It was shown in [19, 20, 24, 25] that when doing a random walk of length chosen randomly between 1 and $O(1/\phi)$ starting at a random node $s \in S$, with constant probability one of the induced $n-1$ cuts has conductance at most $\tilde{O}(\sqrt{\phi})$. In [9], it is observed that the technique still works if the random walk probabilities are only approximately computed and the technique is applied to find low conductance cuts in the streaming model. Based on the streaming algorithm of [9], a simple distributed algorithm to solve the same problem in the CONGEST model (i.e., in a message passing model with messages of logarithmic size [23]) has been presented in [10].

The algorithm of [10] is a straightforward implementation of the ideas of [19, 20, 24, 25] in that for each of the random walks it computes, the complete information to compute the sizes of all $n-1$ induced cuts is sent to a global leader (for each node, one needs to know the number of edges to predecessors/successors in the order given by the random walk probabilities). As a result, for each random walk, the algorithm of [10] requires $O(n)$ rounds to compute the sizes of all cuts induced by the computation of one random walk in G. This results in an overall running time of $\tilde{O}\big(\frac{1}{b}\big(\frac{1}{\phi}+n\big)\big)$ to find a cut of conductance at most $\tilde{O}(\sqrt{\phi})$ and balance at least $\Omega(b)$, where the balance $b \leq 1/2$ of a cut is defined as the ratio of the volume of the smaller side and the total volume of the graph. In this paper we improve on this in the following way. Because it is sufficient to have approximate random walk probabilities, we can round the computed probabilities so that we can partition them $\tilde{O}(1/\phi)$ classes of equal normalized probabilities. Within each class, the nodes can then be ordered arbitrarily. Using a given spanning tree of the network to define the order within each class, we then show that the sizes of all cuts can be computed by doing two appropriate all-prefix-sums computations for each class. We show that this can be done efficiently by developing a distributed variant of the classic parallel all-prefix-sums algorithm [16, 17, 22, 26]. As a result, we obtain an improved running time of $\tilde{O}(D + 1/b\phi)$ for computing a cut of balance $\Omega(b)$ and conductance $\tilde{O}(\sqrt{\phi})$, where D is the diameter of G.

The remainder of the paper is organized as follows. In Section 2, we formally define the communication model and the problem statement. We then formally state the contributions of the paper in Section 3. Sections 4 and 5 are devoted to our technical results, where in Section 4, we first describe the distributed all-prefix-sums algorithm which will then be applied in Section 5, where our main result, an algorithm to compute low conductance cuts will be presented.

2 Model and Definitions

Distributed Computing Model. The network is modeled as an undirected n-node graph $G = (V, E)$. For simplicity, we assume that the graph G is unweighted. We however note that it is not hard to generalize the presented sparse cut approximation algorithm to weighted graphs.[1] We model communication by using the standard CONGEST model: Communication happens in synchronized rounds; in every round, every node is allowed to send a (possibly different) message of at most $O(\log(n))$ bits to each of its neighbors [23]. The *time complexity* of an algorithm is defined as the total number of rounds needed until all nodes terminate. Note that we use the common assumption that local computations at the nodes are for free. We however point out that our algorithms only require very simple, efficient local computations. We further assume that each node is equipped with a unique identifier (ID). Initially, each node knows its own ID as well as the IDs of all its neighbors. Wherever convenient, we slightly abuse notation and identify a node v with its ID.

Random Walks. Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. We use $p_\ell(s, t)$ to denote the probability that a uniform lazy random walk of length ℓ starting at node $s \in V$ ends at node $t \in V$.[2] Hence, $\{p_\ell(s, t) : t \in V\}$ is the probability distribution on nodes induced by a random walk of length ℓ starting from the source node s. For convenience, we abbreviate $p_\ell(s, t)$ by $p(t)$ when source node and length are clear from the text. Let $d(v)$ denote the degree of a node $v \in V$. Given a probability distribution $p(\cdot)$ on the nodes and a node $v \in V$, we further define $\rho_p(v) := p(v)/d(v)$ as the normalized probability of v. Note that the stationary $\pi(\cdot)$ distribution of the uniform lazy random walk is given by $\pi(v) = d(v)/m$ for all $v \in V$.

Cuts and Conductance. Let S be a subset of the nodes V and let $\bar{S} := V \setminus S$. The bipartition $(S, \bar{S})$ of the nodes is called the *cut* induced by S (or also by $\bar{S}$). We use $E(S, \bar{S})$ to denote the set of edges across the cut $(S, \bar{S})$ and $e(S, \bar{S}) := |E(S, \bar{S})|$ for the size of the cut. We measure the *sparsity* of a cut by its *conductance*, where the conductance is defined as follows.

[1] In a weighted graph, we need to substitute the random walk transition matrix with the weighted transition matrix. Also volume and conductance we have to be defined in the natural way for weighted graphs.

[2] In a uniform lazy random walk, in each step, the walk stays at the current node with probability 1/2 and otherwise it moves to a uniformly random neighbor.

▶ **Definition 1** (Conductance). Given a graph $G = (V, E)$, the conductance $\phi(S)$ of the cut $(S, \bar{S})$ induced by a set $S \subseteq V$ is defined as

$$\phi(S) := \frac{e(S, \bar{S})}{\min\left\{\textsc{vol}(S), \textsc{vol}(\bar{S})\right\}},$$

where $\textsc{vol}(S) := \sum_{v \in S} d(v)$ is the *volume* of the node set S. Note that clearly $\phi(\bar{S}) = \phi(S)$. The conductance of the graph G is defined as

$$\phi(G) := \min_{S \subseteq V} \phi(S).$$

A *sparsest cut* of G is a cut $(S, \bar{S})$ with conductance $\phi(S) = \phi(G)$. The performance of our algorithm also depends on the *balance* of the cut it computes.

▶ **Definition 2** (Balance). The balance of a cut $(S, \bar{S})$ is denoted by $b(S) = b(\bar{S})$ and it is defined as

$$b(S) := \frac{\min\left\{\textsc{vol}(S), \textsc{vol}(\bar{S})\right\}}{\textsc{vol}(V)} = \frac{\min\left\{\textsc{vol}(S), \textsc{vol}(\bar{S})\right\}}{2m}.$$

3 Contributions

We develop approximation algorithm for computing sparse cuts in distributed networks. Given two constants b and ϕ, our algorithm outputs a cut of balance at least $b/2$ and conductance at most $\tilde{O}(\sqrt{\phi})$, provided that there is a set $C \subseteq V$ such that $\phi(C) \leq \phi$ and $b(C) \geq b$. Formally, we prove the following main theorem.

▶ **Theorem 3.** *Given a network graph $G = (V, E)$ and two parameters $b \leq 1/2$ and $\phi < 1$ such that there exits a set $C \subseteq V$ with $b \cdot 2|E| \leq \textsc{vol}(C) \leq |E|$ and $\phi(C) \leq \phi$. Then there is a distributed algorithm that finds a cut $(S, \bar{S})$ which satisfies $b|E| \leq \textsc{vol}(S) \leq |E|$ and $\phi(S) = O\big(\sqrt{\phi \log n}\big)$ with high probability and finishes in $O\big(D + \frac{\log^2 n}{b\phi}\big)$ rounds in the CONGEST model, where D is the diameter of G.*

Often, we are most interested in computing an approximation of the *sparsest cut* of the graph G. Assume that there is a sparsest cut (i.e., a cut with conductance $\phi(G)$) with balance b. If $\phi(G)$ and b are known, the above theorem then guarantees to find a cut with conductance $O\big(\sqrt{\phi(G)\log(n)}\big)$ and balance at least $b/2$ in time $O\big(\log^2(n)/(b\phi(G))\big)$ (note that we always have $D = O\big(\log(n)/\phi(G)\big)$). Note that the diameter D clearly is a lower bound on computing any approximation of the sparsest cut. Further, there are graphs with diameter $D = \Theta(\log(n)/\phi(G))$ [8]. Hence, if the sparsest cut has constant balance, this above result is optimal up to a factor $O(\log(n))$ in some graphs. We further mention that the lower bound $\tilde{\Omega}(\sqrt{n} + D)$ for computing sparsest cut shown in [10], only works for weighted graphs. Moreover, the graph they considered to claim the lower bound has very small conductance.

Our algorithm can also be extended to compute the sparsest cut without knowing the conductance value $\phi(G)$. The running time then increases to $\tilde{O}\big(\frac{\tau}{b}\big)$, where τ is the mixing time of the lazy random walk on G. This follows because one can easily estimate τ of G in time $\tilde{O}(\tau)$ and thanks to the relation $\Theta\big(\frac{1}{\phi(G)}\big) \leq \tau \leq \Theta\big(\frac{\log(n)}{\phi(G)^2}\big)$ [14].

In order to obtain the time complexity proven in Theorem 3, we develop a result on the distributed computation of all-prefix-sums which might be of independent interest. Essentially, we show that if the ordering of the nodes can be chosen based on the topology of the network an all-prefix-sums instance where each node has some input can be computed in time $O(D)$. Further, we also show that K independent such all-prefix-sums instances can be evaluated in time $O(D+K)$. For a formal problem statement and the formal results, we refer to Section 4.

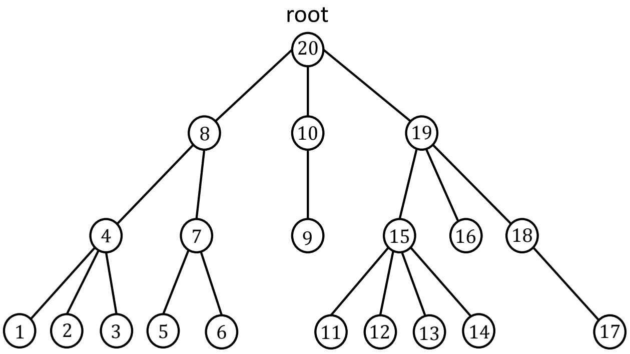


Figure 1 A rooted search tree where the position (in the total order) of each node is greater than the positions of all children and thus the positions of all nodes in its subtrees. The sub-trees are drawn such that they are ordered from left to right.

4 Distributed Prefix Sums Computation

The all-prefix-sums problem takes an ordered tuple of n elements or values $(a_1, a_2, \ldots, a_n)$ and outputs the sums of all prefixes $(a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots \oplus a_n)$ with respect to some binary associative (addition) operation $\oplus$. While computing all prefix sums in (optimal) linear time is a trivial task for ordinary sequential algorithms, the problem is more interesting in a parallel or distributed setting. It is a well-known result that the problem can be solved using logarithmic depth and linear work parallel algorithm in all standard parallel computing models, e.g., on an EREW PRAM [16, 17, 22, 26]. As a result the all-prefix-sums computation is used as a basic building block for many classic parallel algorithms [6].

In the present paper, we adapt the classic parallel all-prefix-sums algorithm to a distributed algorithm in the CONGEST model. As the communication network, we assume that we are given an arbitrary N-node rooted search tree $T = (V_T, E_T)$ with root node $v_r \in V_T$ and radius R (w.r.t. v_r). The search order is defined as follows. At each node $v \in V_T$ with children $u_1, \ldots, u_c$, we are locally given a total order $\prec_v$ on the nodes $\{v, u_1, \ldots, u_c\}$. The overall total order $\prec$ on V_T is then given by combining the local orders $\prec_v$ and by extending the resulting partial order as follows. Given a node u and its parent p, if $u \prec_p p$ (and thus $u \prec p$), we have $w \prec p$ for all nodes w in the subtree of u. Similarly, if $p \prec_p u$ (and thus $p \prec u$), we have $p \prec w$ for all nodes w in the subtree of u. For the remainder of the section, for convenience, we name the nodes in V_T by $v_1, \ldots, v_N$ such that $v_1 \prec v_2 \prec \cdots \prec v_N$. We assume that each node $v \in V_T$ initially knows its parent, all its children, as well as the local order $\prec_v$. For an example of a search tree, see Figure 1.

Each node $v \in V_T$ is given an input value a_v, where a_v is from the domain on which the prefix-sums operation $\oplus$ is defined. We assume that each of the values a_v and also each sum (w.r.t. operation $\oplus$) of a subset of the N input values can be represented using $O(\log N)$ bits. For every $k \in \{1, \ldots, N\}$ and thus for every $v_k \in V_T$, the corresponding k^{th} prefix sum s_{v_k} is defined by $s_{v_1} := a_{v_1}$ and $s_{v_k} := s_{v_{k-1}} \oplus a_{v_k}$ for $k > 1$. In a distributed all-prefix-sums algorithm, each node $v_i \in V_T$ needs to compute the corresponding prefix sum s_{v_i}.

We next present a distributed algorithm to solve the all-prefix-sums problem in T in time $O(R)$ (where R is the radius or depth of T) in the CONGEST model. Note that this time complexity is clearly asymptotically optimal since for every pair of nodes $u, v \in V_T$ with $u \prec v$, the prefix sum of v depends on the value of u and thus the diameter of T is a trivial lower bound on the time complexity to compute all prefix sums.

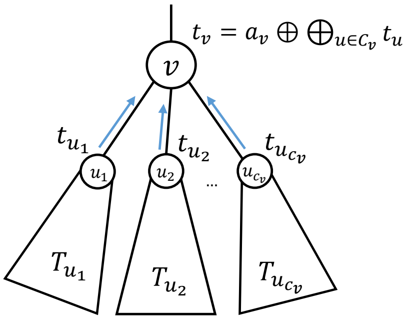


Figure 2 Bottom-up computation by each node v.

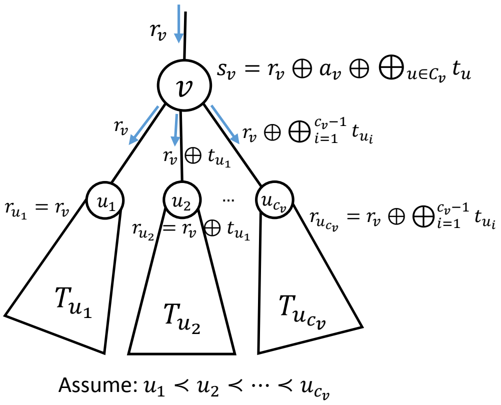

r_v
$s_v = r_v \oplus a_v \oplus \bigoplus_{u \in C_v} t_u$
v
r_v
$r_v \oplus t_{u_1}$
$r_v \oplus \bigoplus_{i=1}^{c_v - 1} t_{u_i}$
$r_{u_1} = r_v$
u_1
u_2
...
u_{c_v}
$r_{u_2} = r_v \oplus t_{u_1}$
$r_{u_{c_v}} = r_v \oplus \bigoplus_{i=1}^{c_v - 1} t_{u_i}$
T_{u_1}
T_{u_2}
$T_{u_{c_v}}$
Assume: $u_1 \prec u_2 \prec \cdots \prec u_{c_v}$

Figure 3 Top-down computation by each node v.

The distributed all-prefix-sums algorithm is an adaptation of the classic parallel algorithm [16, 17, 22, 26] to arbitrary search trees (the classic algorithm builds up a binary search tree). The algorithm consists of two phases. First, there is a bottom-up (convergecast) phase starting from the leaves to compute the sum of the values in each subtree of T. The second phase is a top-down phase in which the prefix sums at all nodes are computed level by level.

Bottom-up Phase (Figure 2). For a node $v \in T$, let T_v be the subtree of T rooted at node v and let $V(T_v)$ be the set of nodes of T_v (including v). Further, for a node v, let C_v be the set of its children and let $c_v := |C_v|$ be the number of its children. We define $t_v := \bigoplus_{u \in T_v} a_u$ to be the sum of all the values in T_v. In the bottom-up phase, each node v recursively computes the value t_v in the obvious way using a convergecast from the leaves to the root, i.e., $t_v = a_v \oplus \bigoplus_{u \in C_v} t_u$.

Top-down Phase (Figure 3). Once the root node v_r has computed t_{v_r}, it initiates the top-down phase. In the top-down phase, each node v computes a value r_v which is defined as follows

$$r_v := \bigoplus_{u \in V_T \setminus V(T_v) : u \prec v} a_u,$$

i.e., r_v is the sum of all input values smaller than v which are not in the subtree of v. After the bottom-up phase, each node $v \in V_T$ knows the value t_v, as well as the values t_u for all children u of v. First note that once a node v also knows r_v, it is straightforward to compute s_v as

$$s_v = r_v \oplus a_v \oplus \bigoplus_{u \in C_v : u \prec v} t_u$$

We can therefore concentrate on the computation of r_v at each node v. For the root node v_r, we clearly have $r_{v_r} = 0$ (here, 0 is assumed to be the neutral element of the operation $\oplus$). For all other nodes $v \in V_T$, let p_v be the parent node of v. As shown by the following lemma, the value of r_v can be computed from r_{p_v}, a_{p_v}, and the values t_u of the children of p_v and of v.

▶ **Lemma 4.** *Let v be a non-root node of T and let p_v be the parent node of v. It holds that*

$$r_v = \begin{cases} r_{p_v} \oplus \bigoplus_{u \in C_{p_v} : u \prec v} t_u & \text{if } v \prec p_v, \\ r_{p_v} \oplus a_{p_v} \oplus \bigoplus_{u \in C_{p_v} : u \prec v} t_u & \text{if } p_v \prec v. \end{cases} \tag{1}$$

Proof. First consider three nodes $u, v, w \in V_T$ such that v is in the subtree of u, but w is not in the subtree of u. Observe that because T is a search tree, it holds that $v \prec w$ if and only if $u \prec w$. For all nodes w which are not in the subtree of p_v it therefore holds the $w \prec p_v$ if and only if $w \prec v$ (and certainly w is also not in the subtree of v). All input values a_w which contribute to r_{p_v} therefore also contribute to r_v and these are the only input values which contribute to r_v and which are not in the subtree of p_v. The value of r_v can therefore be computed by summing r_{p_v} with all values a_x for which $x \prec v$ and where x is in the subtree of p_v but not in the subtree of v. These are exactly the values of all nodes in the subtrees of children u of p_v for which $u \prec v$ and it also includes the value a_{p_v} of p_v if $p_v \prec v$. This proves the lemma. ◀

The following theorem puts everything together and it also shows that it is possible to efficiently solve several concurrent instances of the all-prefix-sums problem.

▶ **Theorem 5.** *Assume that we are given an N-node search tree T and $K \geq 1$ instances of the all-prefix-sums problem on T. That is, each node v has K inputs $a_{v,1}, \dots, a_{v,K}$ and it needs to compute k output values $s_{v,1}, \dots, s_{v,k}$ such that for all $v \in V_T$ and all $i \in \{1, \dots, K\}$, $s_{v,i} = a_{v,i} \oplus \bigoplus_{u \in V_T : u \prec v} a_{u,i}$. In the CONGEST model, the K concurrent all-prefix-sums instances can be computed in time $O(R + K)$, where R is the radius of T.*

Proof. For $K = 1$, the claimed time complexity follows in a straightforward way from the above algorithm description. Both the bottom-up and the top-down phase can clearly be implemented in R rounds. In the bottom-up convergecast, each node v only needs to report t_v to its parent once it knows t_u of all children u. In the top-down phase, as soon as a node u knows r_u, it sends a_u and $\bigoplus_{w \in C_u : w \prec v} t_w$ to each of its children $v \in C_u$. Note that by the assumptions we made, all the messages have a size of at most $O(\log N)$ bits.

For an integer $i \geq 0$, let L_i be the set of nodes at distance exactly i from the root node v_r in T. We call the nodes in L_i the level-i nodes. Note that both the bottom-up and the top-down phase can be implemented such that in each communication round, only the nodes on one level L_i are active. The K concurrent all-prefix-sums instances can therefore be solved in time $O(R + K)$ by using pipelining. ◀

To conclude the section, we adapt the above result to somewhat more general case that we will use for our sparse cut algorithm. As above, assume that as a network, we are given an N-node rooted search tree $T = (V, E)$. However, we will now assume that for the all-prefix-sums problem, the nodes are only partially ordered according to the global order $\prec$ induced by T. Consider a second global order $\sqsubset$ which is defined as follows. We assume that the nodes V are partitioned into K classes $\mathcal{C}_1, \dots, \mathcal{C}_K$. The order $\sqsubset$ is then defined as the lexicographic order define by the class number and the search order $\prec$ of T. That is, for any $i, j \in \{1, \dots, K\}$ and any $u \in \mathcal{C}_i$ and $v \in \mathcal{C}_j$, we have $u \sqsubset v$ if and only if either $i < j$ or $i = j$ and $u \prec v$. Assume that each node $v \in V$ has an input value b_v from the domain for which the associative operator $\oplus$ is defined. We define the prefix sum of node v as $\sigma_v := b_v \oplus \bigoplus_{u \in V : u \sqsubset v} b_u$. The following theorem shows that as long as the number of classes $\mathcal{C}_i$ is not too large, the corresponding all-prefix-sums problem can be computed efficiently in the CONGEST model.

▶ **Theorem 6.** *Let $T = (V, E)$ be a rooted search tree and let $\sqsubset$ be a global order on V defined by a partition $\mathcal{C}_1, \dots, \mathcal{C}_K$ as defined above. Further, assume that every node $v \in V$ has an input value b_v. Then, the prefix sums $\sigma_v = b_v \oplus \bigoplus_{u \sqsubset v} b_u$ for all nodes $v \in V$ can be computed in time $O(R + K)$ in the CONGEST model, where R is the radius of T.*

Proof. For each of the node classes $\mathcal{C}_i$, we first define $S_i := \bigoplus_{v \in \mathcal{C}_i} b_v$ to be the sum of all inputs of nodes in $\mathcal{C}_i$. By doing a standard convergecast on T, for each i, S_i can be computed in R rounds. Also, by using pipelining, all K values $S_1, \dots, S_K$ can be computed in time $R + K$. Using another $R + K$ rounds, we can also make sure that all nodes know all the values $S_1, \dots, S_K$.

Let us now concentrate on a single node class $\mathcal{C}_i$. For each node $v \in \mathcal{C}_i$, we define the local prefix sum $\bar{\sigma}_v$ as $\bar{\sigma}_v := b_v \oplus \bigoplus_{u \in \mathcal{C}_i : u \sqsubset v} b_u$. Note that within a single node class, the two global orders $\sqsubset$ and $\prec$ are identical. Hence, by defining the input to be 0 for all nodes outside $\mathcal{C}_i$, the local prefix sums $\bar{\sigma}_v$ for all nodes $v \in \mathcal{C}_i$ can be computed in time $O(R)$ by using Theorem 5. Also note that computing the local prefix sums for the K different node classes corresponds to K independent all-prefix-sums computations on T and by Theorem 5, it can therefore be done in time $O(R + K)$. Once every node know all values $S_1, \dots, S_K$, as well as its local prefix sum $\bar{\sigma}_v$, it can locally compute its prefix sum σ_v as follows:

$$\forall i \in \{1, \dots, K\} : \forall v \in \mathcal{C}_i : \sigma_v = \bar{\sigma}_v \oplus \bigoplus_{j<i} S_j.$$

This concludes the proof. ◀

5 Algorithm for Sparse Cut

In this section, we present our main result, a distributed algorithm to compute a cut of low conductance. More specifically, we are given an undirected network graph G, a target conductance ϕ, and a balance b as inputs. If there exists a cut $(S, \bar{S})$ with balance at least b and conductance at most ϕ, our distributed algorithm finds a cut $(S', \bar{S'})$ with conductance at most $\tilde{O}\left(\sqrt{\phi}\right)$ and balance at least $b/2$. At the end of the algorithm, every node in G knows whether it is in S' or in $\bar{S'}$. Note that throughout the section, we assume that the number of nodes n is sufficiently large. We can do this w.l.o.g., as if n is a constant, we can always collect the whole graph and compute all cuts in constant time. Throughout the section, we also assume that we have a BFS rooted tree T of the network graph G available. Note that such a tree has depth at most D (where D is the diameter of G) and it can be computed in $O(D)$ rounds in the CONGEST model. We further assume that v_r is the root node of T.

Our algorithm is based on computing probabilities of random walks of multiple lengths ℓ and for multiple sources s. The probabilities of each such random walk define a global order on the nodes V. For the following discussion, we specify a global order on V by a bijection $\pi : V \to \mathbb{N}$ between V and $\{1, \dots, n\}$. That is, a node u appears before v according to the global order π if and only if $\pi(u) < \pi(v)$. Given a global order π on V and an integer $i \in \{1, \dots, n-1\}$ we define the node set $S_\pi(i) := \{v \in V : \pi(v) \leq i\}$. Further, as defined in Section 2, let $p_\ell(s, v)$ be the probability to reach $v \in V$ after exactly ℓ steps of a lazy random walk started at node $s \in V$. Further, recall that for a probability distribution $p(v)$ on the nodes $v \in V$, we use $\rho_p(v) := p(v)/d(v)$ to denote the normalized probability of node v. Our distributed low conductance cut algorithm uses conductance approximation techniques developed by Lovász and Simonovits [19, 20] and by Spielman and Teng [24, 25]. Formally, we apply the following lemma which is a relatively simple application of the results of [19, 20, 24, 25] and which was formally proven in [9].

▶ **Lemma 7** ([9]). *Let $G = (V, E)$ be a graph and let $(S, \bar{S})$ be a cut of G of conductance at most ϕ such that $\mathrm{VOL}(S) \le \mathrm{VOL}(V)/2$. Further, let $s \in S$ be a node sampled randomly from the degree distribution in S and let ℓ be an integer chosen uniformly at random from $\{1, \dots, 1/8\phi\}$. We define $p(v) := p_\ell(s, v)$ and we assume that for all $v \in V$, $\tilde{p}(v)$ is an estimate for the probability $p(v)$ such that $|\tilde{p}(v) - p(v)| \le \frac{\epsilon}{2}(p(v) + 1/n)$, where $\epsilon < \phi$. Let $\pi : V \to \mathbb{N}$ be any global order on V such that $\pi(u) < \pi(v)$ whenever $\rho_{\tilde{p}}(u) > \rho_{\tilde{p}}(v)$. Then with constant probability for some set $S_\pi(i)$ for $i \in \{1, \dots, n-1\}$, we have $\phi(S_\pi(i)) \le 8\sqrt{\phi \log(n)}$ and $b(S_\pi(i)) \ge b(S)/2$.*

Based on Lemma 7, the strategy for computing a cut of low conductance is as follows. Assume that we are given a network graph $G = (V, E)$ and two parameters $2/n^2 \le \phi < 1$ and $b \le 1/2$. For a sufficiently large constant[3] $c > 0$, we define a parameter $Q = \frac{c \cdot \ln n}{b}$. We randomly (independently) select Q nodes $s_1, \dots, s_Q$ and Q lengths $\ell_1, \dots, \ell_Q$, where each node s_i is chosen according to the degree distribution of G and each length ℓ_i is chosen uniformly from the range $\{1, \dots, 1/8\phi\}$. For each $i \in \{1, \dots, Q\}$, the approximate random walk probabilities $\tilde{p}(v)$ for a walk of length ℓ_i starting at s_i are computed. It then follows directly from Lemma 7 that if c is chosen sufficiently large and if the graph G has a cut $(S, \bar{S})$ with $\phi(S) \le \phi$ and $b(S) \ge b$, with high probability, for at least one of the Q random walks, one of the computed $n - 1$ cuts has the desired balance and conductance.

The core of our distributed sparse cut algorithm therefore is to compute approximate random walk probabilities for a given starting node s and a given length ℓ and to compute the conductances and balances of the $n - 1$ cuts induced by these approximate random walk probabilities. Theorem 3 will then follow by repeating this $O(\log(n)/b)$ times. In the following, we therefore assume that we have a fixed start node $s \in V$ and a fixed random walk length $\ell \le 1/8\phi$. We will first show how to compute approximate probabilities $\tilde{p}(v) \approx p_\ell(s, v)$. As a second step, we will show that the properties of these probabilities $\tilde{p}(v)$ allow to use the all-prefix-sums result of Section 4 to quickly compute the balances and conductances of the induced cuts.

5.1 Computing the Random Walk Probabilities

We estimate the probability distribution of a random walk starting from a starting node $s \in V$. Recall that we perform a lazy random walk, i.e., in each step, the walk stays at the current node with probability 1/2. The probability of $p_t(s, v)$ of being at node v after t steps of the random walk can be stated recursively as follows. For $t = 0$, $p_0(s, s) = 1$ and $p_0(s, v) = 0$ for any $v \neq s$. For $t > 0$, we have

$$p_t(s, v) = \frac{1}{2} \cdot p_{t-1}(s, v) + \sum_{u \in N(v)} \frac{p_{t-1}(s, u)}{2d(u)}. \tag{2}$$

Hence, given $p_{t-1}(s, v)$ for all nodes v, in principle, it is possible to exactly compute $p_t(s, v)$ for all nodes v in a single communication round. Note however that the probabilities $p_t(s, v)$ are real values and since in the CONGEST model we are restricted to using at most $O(\log(n))$ bits per message, we need to be a little bit more careful. In the following, assume that for each time t, each node v maintains an approximation $\beta_t(v)$ of $p_t(s, v)$ and let $\boldsymbol{\beta_t}$ be the n-vector of all these approximations. We define $\delta_t(v) := \beta_t(v) - p_t(s, v)$ to be the error of v's approximation

[3] The constant only helps to measure the high probability bound of the result; larger the constant value means higher the probability guarantee.

of $p_t(s,v)$ and we use $\boldsymbol{\delta_t}$ to denote the vector of all errors after step t of the random walk. We assume that the approximations $\beta_t(v)$ are computed as follows. Node v collects $\beta_{t-1}(u)$ from all neighbors u, it evaluates $\beta'_t(v) := \beta_{t-1}(v)/2 + \sum_{u \in N(v)} \beta_{t-1}(u)/2d(u)$, and it then computes $\beta_t(v)$ as $\beta'_t(v)$ rounded to the closest integer multiple of n^8. Note that because $\beta_t(v)$ is always between 0 and 1, there are at most n^8+1 different values for $\beta_t(v)$ and therefore all messages can clearly be encoded using $O(\log(n))$ bits. The following lemma shows that also after ℓ steps, the absolute error $|\delta_\ell(v)|$ of all nodes v is still small.

▶ **Lemma 8.** *For all $v \in V$ and $t \geq 0$, we have $|\delta_t(v)| = |\beta_t(v) - p_t(s,v)| \leq t \cdot n^{-8}$.*

Proof. We prove the lemma by induction on t. As $\beta_0(v) = p_0(s,v)$, the lemma is true for $t=0$. Let us therefore consider the induction step. Let T be the transition matrix of the considered lazy random walk on G. The recursion 2 can then be expressed as $\boldsymbol{p_t} = T \cdot \boldsymbol{p_{t-1}}$, where $\boldsymbol{p_t}$ is the n-vector defined by the probabilities $p_t(s,v)$ for each node v. Similarly, the vector $\boldsymbol{\beta'_t}$ is computed as

$$\boldsymbol{\beta'_t} = T \cdot \boldsymbol{\beta_{t-1}} = T \cdot (\boldsymbol{p_{t-1}} + \boldsymbol{\delta_{t-1}}) = \boldsymbol{p_t} + T \cdot \boldsymbol{\delta_{t-1}}.$$

Note that because T is a stochastic matrix, $T \cdot \boldsymbol{\delta_{t-1}}$ is a convex combination of the values $\delta_{t-1}(u)$ for $u \in \{v\} \cup N(v)$. The absolute value of $T \cdot \boldsymbol{\delta_{t-1}}$ can therefore be upper bounded by the largest absolute value of $\delta_{t-1}(u)$ for any $u \in V$. By induction, we therefore have $|\beta'_t(v) - p_t(s,v)| \leq (t-1)n^{-8}$. The lemma now follows because $|\beta_t(v) - \beta'_t(v)| \leq n^{-8}$. ◀

We next define how the estimates $\tilde{p}(v) \approx p_\ell(s,v)$ are computed. For sufficiently large constant c, the estimates $\beta_\ell(v)$ are accurate enough to be used in Lemma 7. However, in order to efficiently compute the conductances and balances of all cuts induced by the global order given by the probabilities $\tilde{p}(v)$, we will apply the Theorem 6 (on computing all-prefix-sums). In Theorem 6, we would like the number of node classes to be as small as possible. We therefore define $\delta := \phi/10$ and $\tilde{p}(v)$ as follows:

$$\forall v \in V : \tilde{p}(v) := \begin{cases} 0 & \text{if } \beta_\ell(v) \leq n^{-6}, \\ d(v) \cdot (1+\delta)^{\left\lfloor \log_{1+\delta}\left(\frac{\beta_\ell(v)}{d(v)}\right)\right\rfloor} & \text{otherwise.} \end{cases} \tag{3}$$

That is, $\tilde{p}(v)$ is either 0 or we round down $\beta_\ell(v)/d(v)$ to the next smaller power of $1+\delta$ and we multiply the resulting value by $d(v)$. This guarantees that the value of $\rho_{\tilde{p}}(v) = \tilde{p}(v)/d(v)$ is equal to $\beta_\ell(v)/d(v)$ rounded to the next smaller power of $1+\delta$.

▶ **Lemma 9.** *For all $v \in V$, it holds that $|\tilde{p}(v) - p_\ell(s,v)| \leq \delta\left(n^{-3} + p_\ell(s,v)\right)$. Further, the value $\rho_{\tilde{p}}(v) = \tilde{p}(v)/d(v)$ can only have $O(\log(n)/\phi)$ different values.*

Proof. The second claim follows because $\rho_{\tilde{p}}(v)$ is always a value between 0 and 1 and because it either is 0 or it is of size at least $\Omega(n^{-5})$ and it is an integer power of $1+\delta = 1+\phi/10$.

For the first claim, first observe that because $\phi \geq 2/n^2$, we always have $\ell \leq n^2/16$. Lemma 8 therefore implies that $|p_\ell(s,v) - \beta_\ell(v)| \leq n^{-6}/16$.

Let us first consider the case $\tilde{p}(v) = 0$. In this case, from Equation (3) and Lemma 8, we then get that $p_\ell(s,v) < 2n^{-6}$ and for sufficiently large n, the lemma follows because $\delta = \phi/10 \geq 5/n^2$.

Let us therefore assume that $\tilde{p}(v) > 0$. By the definition of $\tilde{p}(v)$, we then have $1 \leq \beta_\ell(v)/\tilde{p}(v) \leq 1+\delta$. We again use that $|\beta_\ell(v) - p_\ell(s,v)| \leq n^{-6}/16$. Using $\tilde{p}(v) \leq \beta_\ell(v)$, we get that $\tilde{p}(v) \leq p_\ell(s,v) + n^{-6}/16$. Further, by using that $\beta_\ell(v) \leq (1+\delta)\tilde{p}(v) \leq \tilde{p}(v) + \delta\beta_\ell(v)$, we have $(1-\delta)(p_\ell(s,v) - n^{-6}/16) \leq (1-\delta)\beta_\ell(v) \leq \tilde{p}(v)$ and therefore $\tilde{p}(v) \leq p_\ell(s,v) + \delta(p_\ell(s,v) + n^{-6}/16)$. ◀

▶ **Lemma 10.** *Assume that we compute the probability estimates $\tilde{p}(v)$ for Q different random walks where each random walk is started at a random node s chosen according to the degree distribution of G and the length of each random walk is chosen uniformly at random from $\{1, \dots, 1/8\phi\}$. The probability estimates $\tilde{p}(v)$ for all Q random walks can be computed in $O(D + Q/\phi)$ rounds in the CONGEST model, where D is the diameter of the G.*

Proof. Let us first consider the computation of a single random walk. As a first step, we need to randomly choose the starting node s and the length ℓ of the random walk. We can use the BFS tree T to do this. The length ℓ of the random walk can be determined by the root node v_r of T and it can be sent to all nodes in at most D rounds. Assume that each node in T knows the sum of the degrees of all nodes in its subtree. This information can be computed by a simple convergecast in D rounds. Further, based on this knowledge, we can now choose the starting node of the random walk by randomly walking down the tree starting at v_r (at each node we stop or go to a subtree with probability proportional to the corresponding degree sum). The node on which the random walk stops will be the sampled node according to the degree distribution of G. Also note that by using pipelining, the starting nodes and lengths of all the Q random walks can be computed in time $O(D + Q)$.

To compute the probability estimates $\tilde{p}(v)$ of a single random walk, it directly follows from the above discussion that $\beta_\ell(v)$ can be computed in ℓ rounds. Given $\beta_\ell(v)$, node v can compute $\tilde{p}(v)$ locally without any further communication. The lemma therefore follows because $\ell = O(1/\phi)$. ◀

5.2 Evaluating the Induced Cuts

Consider the probability estimates $\tilde{p}(v)$ for some random walk of length ℓ with starting node $s \in V$. We assume that the estimate $\tilde{p}(v)$ are computed as described above. By Lemma 9, the estimates are accurate enough to be used in Lemma 7. In order to evaluate the conductances and balances of the cuts induced by the probability estimates $\tilde{p}(v)$, we intend to use the all-prefix-sums techniques developed in Section 4. In order to apply these techniques, we need a distributed search tree. For this purpose, we can again use the computed BFS tree T of G. In order to get a search tree from T, every node just needs to arbitrarily order its children. Note that the radius of T is upper bounded by the diameter D of G. Let $\prec$ be the search order (on V) defined by the search tree T.

Given the probability estimates $\tilde{p}(v)$, we define $\rho_{\tilde{p}} := \tilde{p}(v)/d(v)$ as before. In order to apply Lemma 7, we need to define a global order $\pi : V \to \mathbb{N}$ on V such that $\pi(u) < \pi(v)$ whenever $\rho_{\tilde{p}}(u) > \rho_{\tilde{p}}(v)$. We define this global order π such that $\pi(u) < \pi(v)$ if and only if either $\rho_{\tilde{p}}(u) > \rho_{\tilde{p}}(v)$ or $\rho_{\tilde{p}}(u) = \rho_{\tilde{p}}(v)$ and $u \prec v$. We first show that an all-prefix-sums problem w.r.t. this global order can be computed efficiently in the CONGEST model.

▶ **Lemma 11.** *Assume that each node $v \in V$ has an integer value b_v (of size at most polynomial in n). Further assume that each node $v \in V$ needs to compute $s_v := \sum_{u \in V : \pi(u) \leq \pi(v)} b_u$. The values s_v for all $v \in V$ can be computed in $O(D + \log(n)/\phi)$ rounds in the CONGEST model. Further, the results of K independent such all-prefix-sums problems (possibly for different random walk probabilities) can be computed in time $O(D + K \log(n)/\phi)$ in the CONGEST model.*

Proof. We first prove the lemma for a single instance of the described all-prefix-sums instance. Note that the global order defined by π has the structure of the order $\sqsubset$ used in Theorem 6. All nodes v with equal value $\rho_{\tilde{p}}(v)$ form a single node class. Two nodes u and v of different classes are then order by the values of $\rho_{\tilde{p}}(u)$ and $\rho_{\tilde{p}}(v)$. Two nodes u and v in the same class are order by the search tree order $\prec$. We can therefore directly apply Theorem 6 to compute the values s_v for all nodes v. The time complexity of doing this is $O(n+k)$, where k is the number of different node classes. It follows from the second claim of Lemma 9 that the number of classes is at most $O(\log(n)/\phi)$ and thus the lemma follows for $K = 1$.

For $K > 1$, note that we can use pipelining as described in Theorem 5. For each instance of the all-prefix-sums problem, we run $O(\log(n)/\phi)$ independent all-prefix-sums instances w.r.t. the search tree order $\prec$. In total, we therefore run $O(K \log(n)/\phi)$ independent simple all-prefix-sums instances and the claim of the lemma thus follows. ◀

It remains to show that the problem of evaluating the cuts for a given total order $\pi : V \to \mathbb{N}$ on V can be reduced to computing a few all-prefix-sums computations. Let us therefore consider a global order π on V. We need to compute the conductances and balances of all the cuts defined by the sets $S_\pi(i)$ for $i \in \{1, \dots, n-1\}$. For a node $v \in V$, we define $d^+(v) := |\{u \in N(v) : \pi(u) > \pi(v)\}|$ and let $d^-(v) := |\{u \in N(v) : \pi(u) < \pi(v)\}|$. Note that $d^+(v) + d^-(v) = d(v)$. Note also that every node $v \in V$ can compute $d^+(v)$ and $d^-(v)$ using a single communication round (based on the relative ordering w.r.t. its neighbors). For a node set $S \subset V$, we further let $e(S)$ be the number of edges crossing the cut $(S, \bar{S})$. Recall that $\phi(S) = e(S)/(2mb(S))$ and $b(S) = \min\left\{\textsc{vol}(S), \textsc{vol}(\bar{S})\right\}/2m$. The following lemma shows that the conductances and balances of all cuts $(S_\pi(i), \bar{S}_\pi(i))$ can be reduced to two all-prefix-sums computations w.r.t. the order π.

▶ **Lemma 12.** *We have $e(S_\pi(1)) = \textsc{vol}(S_\pi(1)) = d(v_1)$. For $i > 1$, it further holds that*

$$\begin{aligned} e(S_\pi(i)) &= e(S_\pi(i-1)) + d^+(v_i) - d^-(v_i), \\ \textsc{vol}(S_\pi(i)) &= \textsc{vol}(S_\pi(i-1)) + d(v_i). \end{aligned}$$

Proof. We first consider the first recursion specifying $e(S_\pi(i))$ The set of edges connecting nodes in $S_\pi(i)$ with nodes in $V \setminus S_\pi(i)$ consists of all edges connecting nodes in $S_\pi(i-1)$ with nodes in $V \setminus S_\pi(i)$ and of all edges connecting v_i with nodes in $V \setminus S_\pi(i)$. The number of edges connecting nodes in $S_\pi(i-1)$ with nodes in $V \setminus S_\pi(i)$ is $e(S_\pi(i-1)) - d^-(v_i)$ and the number of edges connecting v_i with nodes in $V \setminus S_\pi(i)$ is $d^+(v_i)$. The first recursion therefore follows. The second recursion follows immediately by the definition of the volume $\textsc{vol}(S)$ of a node set S. ◀

We now have everything we need to prove the main theorem.

▶ **Theorem 3 (restated).** *Given a network graph $G = (V, E)$ and two parameters $b \le 1/2$ and $\phi < 1$ such that there exits a set $C \subseteq V$ with $b \cdot 2|E| \le \textsc{vol}(C) \le |E|$ and $\phi(C) \le \phi$. Then there is a distributed algorithm that finds a cut $(S, \bar{S})$ which satisfies $b|E| \le \textsc{vol}(S) \le |E|$ and $\phi(S) = O\big(\sqrt{\phi \log n}\big)$ with high probability and finishes in $O\big(D + \frac{\log^2 n}{b\phi}\big)$ rounds in the CONGEST model, where D is the diameter of G.*

Proof. We have already seen that Lemma 7 implies the quality of the returned cut with high probability if we consider all the cuts induced by $O(\log(n)/b)$ random walks (where starting node and length of each random walk are chosen randomly as specified by Lemma 7 and the computed probability estimates satisfy the accuracy demanded by Lemma 7). By Lemma 10, the probability estimates $\tilde{p}(v)$ for $O(\log(n)/b)$ such random walks can be computed in $O(D + \log(n)/(b\phi))$ rounds. Further by Lemma 9, the accuracy of the probability estimates $\tilde{p}(v)$ is good enough to be used in Lemma 7.

In order to prove the theorem, it therefore remains to show that for the $O(\log(n)/b)$ random walks, the conductances and balances of all induced cuts can be computed in

$O\big(D + \frac{\log^2 n}{b\phi}\big)$ rounds. By Lemma 12, for a given global order π on the nodes V, the conductances and balances of all cuts $S_\pi(i)$ for $i \in \{1, \ldots, n-1\}$ can be computed by using 2 all-prefix-sums computations (one for $e(S_\pi(i))$ and one for $\textsc{vol}(S_\pi(i))$). The conductances and balances of the cuts of all $O(\log(n)/b)$ random walks can therefore be computed by carrying out $O(\log(n)/b)$ independent all-prefix-sums computations. By Lemma 11, we can therefore compute the conductances and balances of all cuts induced by all random walks in time $O\big(D + \log^2(n)/b\phi\big)$. Note that when doing this, for each cut, possibly only one node in G knows the results. However, we can do a convergecast on the BFS tree T to find the best of all the computed cuts in D additional rounds. This completes the proof of the main theorem. ◀

References

1 Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Using pagerank to locally partition a graph. *Internet Mathematics*, 4(1):35–64, 2007.

2 Reid Andersen and Yuval Peres. Finding sparse cuts locally using evolving sets. In *Proc. of 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 235–244, 2009.

3 Sanjeev Arora and Satyen Kale. A combinatorial, primal-dual approach to semidefinite programs. In *Proc. of 39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 227–236, 2007.

4 Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2), 2009.

5 Sandeep N. Bhatt and Frank T. Leighton. A framework for solving VLSI graph layout problems. *J. Comput. Syst. Sci.*, 28(2):300–343, 1984.

6 Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

7 Keren Censor-Hillel and Hadas Shachnai. Fast information spreading in graphs with large weak conductance. *SIAM J. Comput.*, 41(6):1451–1465, 2012.

8 Flavio Chierichetti, Silvio Lattanzi, and Alessandro Panconesi. Almost tight bounds for rumour spreading with conductance. In *Proc. of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 399–408, 2010.

9 Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Sparse cut projections in graph streams. In *Proc. of 17th Annual European Symposium (ESA)*, pages 480–491, 2009.

10 Atish Das Sarma, Anisur R. Molla, and Gopal Pandurangan. Distributed computation of sparse cuts via random walks. In *Proc. of 16th International Conference on Distributed Computing and Networking (ICDCN)*, pages 6:1–6:10, 2015.

11 Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. Distributed random walks. *J. ACM*, 60(1):2, 2013.

12 George Giakkoupis. Tight bounds for rumor spreading in graphs of a given conductance. In *Proc. Int. Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 57–68, 2011.

13 Christos Gkantsidis, Milena Mihail, and Amin Saberi. Conductance and congestion in power-law graphs. In *Proc. of International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 148–159, 2003.

14 Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM J. Comput.*, 18(6):1149–1178, 1989.

15 David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.

16 Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, 1973.

17 Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

18 Frank T. Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

19 László Lovász and Miklós Simonovits. The mixing rate of markov chains, an isoperimetric inequality, and computing the volume. In *Proc. of 31st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 346–354, 1990.

20 László Lovász and Miklós Simonovits. Random walks in a convex body and an improved volume algorithm. *Random Struct. Algorithms*, 4(4):359–412, 1993.

21 David W. Matula and Farhad Shahrokhi. Sparsest cuts and bottlenecks in graphs. *Discrete Applied Mathematics*, 27(1-2):113–123, 1990.

22 Yu Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics Doklady*, 7(7):589–591, 1963.

23 David Peleg. *Distributed computing: a locality-sensitive approach.* SIAM, Philadelphia, PA, USA, 2000.

24 Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proc. of 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 81–90, 2004.

25 Daniel A. Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, arXiv: abs/0809.3232v1, 2008.

26 Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathe- matical Software*, 1(4):289–307, 1975.

Distributed Approximation of k-Service Assignment

Magnús M. Halldórsson[*1], **Sven Köhler**[†2], **and Dror Rawitz**[2]

1 **Reykjavik University, Reykjavik, Iceland**
`mmh@ru.is`
2 **Bar-Ilan University, Ramat-Gan, Israel**
`sven.kohler@biu.ac.il`
3 **Bar-Ilan University, Ramat-Gan, Israel**
`dror.rawitz@biu.ac.il`

Abstract

We consider the k-Service Assignment problem (k-SA), defined as follows. The input consists of a network that contains servers and clients, and an integer k. Each server has a finite capacity, and each client is associated with a demand and a profit. A feasible solution is an assignment of clients to neighboring servers such that (i) the total demand assigned to a server is at most its capacity, and (ii) a client is assigned either to k servers or to none. The profit of an assignment is the total profit of clients that are assigned to k servers, and the goal is to find a maximum profit assignment. In the r-restricted version of k-SA, no client requires more than an r-fraction of the capacity of any adjacent server. The k-SA problem is motivated by backup placement in networks and by resource allocation in 4G cellular networks. It can also be viewed as machine scheduling on related machines with assignment restrictions.

We present a centralized polynomial time greedy $\frac{k+1-r}{1-r}$-approximation algorithm for r-restricted k-SA. We then show that a variant of this algorithm achieves an approximation ratio of $k+1$ using a resource augmentation factor of $1+r$. We use the latter to present a $(k+1)^2$-approximation algorithm for k-SA. In the distributed setting, we present: (i) a $(1+\varepsilon)\frac{k+1-r}{1-r}$-approximation algorithm for r-restricted k-SA, (ii) a $(1+\varepsilon)(k+1)$-approximation algorithm that uses a resource augmentation factor of $1+r$ for r-restricted k-SA, both for any constant $\varepsilon > 0$, and (iii) an $O(k^2)$-approximation algorithm for k-SA (in expectation). The three distributed algorithms compute a solution with high probability and terminate in $O(k^2 \cdot \log^3 n)$ rounds.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases approximation algorithms, distributed algorithms, related machines

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.11

1 Introduction

We consider the k-Service Assignment problem (abbreviated k-SA). A k-SA instance consists of a set of servers and a set of clients. Each server has a finite capacity, and each client has a demand and a profit. (The demand of a client does not depend on the identity of the server.) A feasible solution is a *k-service assignment* of clients to servers such that:

- A client is only assigned to neighboring servers.
- The total demand of clients that are assigned to a server does not exceed its capacity.
- Each client is assigned either to k servers or to none.

[*] Supported in part by Icelandic Research Fund (grants No. 120032011 and 152679-051).
[†] Supported in part by a grant from the Israeli Ministry of Science, Technology, and Space and by the Israel Science Foundation (grant No. 497/14).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 11; pp. 11:1–11:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A client that is assigned to k servers is said to be *satisfied*, and the profit of a service assignment is the total profit of satisfied clients. The goal in k-SA is to find a service assignment with maximum profit.

Given a constant $r \in (0, 1]$, an instance of k-SA is said to be r-restricted if no client requires more than an r-fraction of the capacity of any neighboring server. k-SA on r-restricted instances is referred to as r-restricted k-SA.

k-SA is NP-hard, since the special case with exactly k servers is equivalent to the KNAPSACK problem. Since KNAPSACK remains NP-hard even if the size of each item is at most an r-fraction of the knapsack size, this hardness result applies to r-restricted k-SA, for any $r \in (0, 1]$. (This was explicitly shown for 1-SA in [1].) This also means that the approximation ratio of the natural greedy algorithm is $\Omega(\frac{1}{1-r})$, even for r-restricted 1-SA.

The k-SA problem naturally arises in network applications where clients need service from (multiple) servers. Amzallag et al. [1] used 1-SA to model the problem of assigning clients to base stations in 4G cellular networks where services offered by providers (such as video streaming and web browsing) require high bit-rates, and client diversity is an issue. By using 1-SA they took into account both base stations diversity (using non-uniform capacities), as well as clients diversity (using different demands, profits, and potential set of base stations). Amzallag et al. [1] also considered the variant of 1-SA where a client c may be serviced by multiple servers as long as the total service it receives is $d(c)$. Such an assignment is called a *cover by many*, while a solution that assigns a single server to a client is called a *cover by one*. They presented a $\frac{2-r}{1-r}$-approximation algorithm that computes *covers by one* and a $\frac{1}{1-r}$-approximation algorithm that computes *covers by many*. In fact the former ratio is in comparison to an optimal *cover by many*. Both algorithms are based on the local ratio technique [5, 3, 4].

Patt-Shamir et al. [18] presented a distributed implementation of the first algorithm from [1] while paying a $(1 + \varepsilon)$ factor in the approximation ratio. That is, they presented a distributed $(1 + \varepsilon)\frac{2-r}{1-r}$-approximation algorithm, for any $\varepsilon > 0$, for r-restricted 1-SA, for any $r \in (0, 1)$. The algorithm requires a polylogarithmic number of rounds in the CONGEST model. The above result is based on two assumptions: (i) the *cost-effectiveness* of clients is polynomially bounded, and (ii) each server knows the demands and profits of adjacent clients. (The definition of cost-effectiveness and a more detailed description of these assumptions are given in the next section.)

Recently, Halldórsson at al. [13] considered the BACKUP LOCATION problem in which each client has a file whose backup should be stored in k neighbors to increase fault tolerance. They mainly focused on the dual problem, where an instance is similar to a k-SA instance and the goal is to satisfy all clients while minimizing the maximum load. They also observed that k-SA is APX-hard, for $k \geq 3$, and showed a lower bound of $\Omega(\frac{k}{\log k})$ for the approximation ratio based on a reduction from k-DIMENSIONAL MATCHING [14].

Our Results. We generalize the $\frac{2-r}{1-r}$-approximation algorithm from [1] by presenting a $\frac{k+1-r}{1-r}$-approximation algorithm for r-restricted k-SA, for any $r \in (0, 1)$. We provide a simplified analysis that does not rely on the local ratio technique. We show that a variant of the above algorithm achieves an approximation ratio of $k + 1$ for r-restricted k-SA, for any $r \in (0, 1]$, using a resource augmentation factor of $1 + r$. Then, by showing that the clients that receive service in the resource augmented solution can be $(k + 1)$-colored, such that each color induces a feasible solution, we obtain a $(k + 1)^2$-approximation algorithm for k-SA. The algorithm outperforms the $\frac{k+1-r}{1-r}$-approximation algorithm, when $r > \frac{k+1}{k+2}$.

Based on the approach taken in [18], we design a distributed version of the former algorithm that, for any constant $\varepsilon > 0$, computes $(1 + \varepsilon)\frac{k+1-r}{1-r}$-approximate solutions for

r-restricted k-SA with high probability and whose running time is $O(k^2 \cdot \log^3 n)$ rounds in the Congest model. While the algorithm for 1-SA from [18] is based on computing a maximal matching, our algorithm is based on computing a maximal packing of stars, where each star consists of a client and k adjacent servers. As in the centralized setting we provide an algorithm that achieves a factor of $(1+\varepsilon)(k+1)$ using a resource augmentation factor $1+r$. We use distributed random selection instead of coloring to design a distributed algorithm for k-SA that computes solutions whose expected profit is a $\Omega(k^{-2})$-fraction of the optimum, using $O(k^2 \cdot \log^3 n)$ rounds. When $k = O(1)$, this amounts to an $O(1)$-approximation algorithm that terminates in $O(\log^3 n)$ rounds.

The results of this paper can be extended to a natural variant of k-SA in which each client $c \in C$ requires service from $k(c) \in \mathbb{N}$ servers. Then $k_{\max} \triangleq \max_{c \in C} k(c)$ replaces k in the approximation ratios and time complexities. We will address this variant in the full version of the paper.

Related Work. 1-SA is equivalent to Multiple Knapsack with Assignment Restrictions (MKAR), where the input consists of a set of bins and a set of items. Each bin has a capacity, and each item j has a size, a profit, and a subset of bins in which it can be placed. A feasible solution is an assignment of items to bins such that each item is assigned to one of the bins in its subset and the total size of items assigned to each bin is at most its capacity. The goal is to find a solution of maximum profit. A special case of MKAR, where the size and profit of an item are the same, was considered by Dawande et al. [8]. They presented an LP-rounding 2-approximation algorithm, a $(2+\varepsilon)$-approximation algorithm that uses an FPTAS for solving a single knapsack problem, and a greedy 3-approximation algorithm.

Fleischer et al. [11] studied the Separable Assignment Problem (SAP). In this problem the input consists of a set of bins and a set of items, and a profit f_{ij} for assigning item j to bin i. There is also a separate packing constraint for each bin, i.e., a collection $\mathcal{I}_i$ of subsets of items that fit in bin i. The goal is to maximize the total profit. Given an α-approximation algorithm for the single machine version of SAP, they presented an LP-rounding based $\frac{\alpha e}{e-1}$-approximation algorithm and a local search $(\frac{\alpha+1}{\alpha} + \varepsilon)$-approximation algorithm, for any $\varepsilon > 0$. If the single machine version admits a PTAS (FPTAS), then the ratios are $\frac{e}{e-1} + \varepsilon$ $(\frac{e}{e-1})$ and $2 + \varepsilon$.

In the Generalized Assignment Problem (GAP) the input consists of a set of bins and a set of items. Each bin has a capacity, and each item j has a size and a profit for each bin i. A feasible solution is an assignment of items to bins such that the total size of items that are assigned to a bin is at most its capacity. GAP is a special case of SAP where the simple knapsack version admits an FPTAS, and thus it has a $\frac{e}{e-1}$-approximation algorithm. MKAR (and hence 1-SA) is a special case of GAP. Chekuri and Khanna [7] gave a PTAS for Multiple Knapsack (without assignment restrictions) and showed that GAP is APX-hard. In addition they observed that an LP-rounding 2-approximation algorithm for the minimization version of GAP by Shmoys and Tardos [22] implies a 2-approximation algorithm for GAP. This result applies to 1-SA.

Amzallag et al. [1] showed that the version of 1-SA that allows *cover by many* cannot be approximated to within a factor which is better than $|J|^{1-\varepsilon}$, for any $\varepsilon > 0$, unless NP=ZPP.

k-SA is a special case of the Packing Integer Programs problem (PIP). In this problem we are given a set of items and a collection of knapsack constraints over these items. The goal is to maximize the profit of packed items. In k-SA each item appears in k constraints with the same coefficient. The single constraint (or server) case is the Knapsack problem which has an FPTAS [21, 15], and the constant number of constraints case is the Multi-dimensional Knapsack problem that has a PTAS [12], while obtaining an FPTAS is NP-hard [17]. Raghavan and Thompson [20] used randomized LP-rounding to obtain an approximation ratio of $O(m^r)$ for PIP, where m is the number of constraints, k is the maximum number of constraints per item, and r is the maximum item coefficient per constraint RHS. Srinivasan [23] improved this ratio to $O(m^{r/(r+1)})$. In k-SA this translates to an $O(|S|^{r/(r+1)})$ ratio, where S is the set of servers. Chekuri and Khanna [6] proved that the above ratio is almost tight by showing that, for every fixed integer α and fixed $\varepsilon > 0$, the special case of PIP where all constraints are composed of binary coefficients and RHS α cannot be approximated within a factor of $m^{1/(\alpha+1)-\varepsilon}$, unless NP=ZPP. They also showed that PIP with uniform RHS α cannot be approximated within a factor of $m^{1/(\alpha+1)-\varepsilon}$, unless NP=ZPP, even with a resource augmentation factor α. Note that this does not contradict our results, since we assume that each item appears in at most k constraints.

Paper Organization. We formally define the problem and the execution model in Section 2. This section also contains definitions and notation that is used in the paper. The centralized algorithms are given and Section 3 and the distributed algorithms are presented in Section 4.

2 Preliminaries

This section contains a formal problem statement, several definitions and notation that we use throughout the paper, and the execution model.

Problem. We consider the k-Service Assignment (k-SA) problem. A k-SA instance consists of a bipartite graph $G = (C, S, E)$, where C is a set of clients and S is a set of servers. Each server $s \in S$ has a positive capacity $cap(s)$, and each client $c \in C$ has a demand $d(c)$ and a profit $p(c)$. We define $n \triangleq |C| + |S|$. A feasible solution is a k*-service assignment* (or simply a *service assignment*) of clients to servers, i.e., it is a function $x : C \times S \to \{0,1\}$ such that:

- A client is only assigned to neighboring servers, namely $x(c,s) = 1$ implies $(c,s) \in E$.
- The total demand of clients assigned to a server is not larger than its capacity, i.e., $\sum_{c \in C} x(c,s) \cdot d(c) \leq cap(s)$, for every server $s \in S$.
- Each client is assigned either to k servers or to none. That is, $\sum_{s \in S} x(c,s) \in \{0,k\}$, for every client $c \in C$.

Given a k-service assignment x, a client is *satisfied* if $\sum_{s \in S} x(c,s) = k$. The set of satisfied clients is denoted by C_x, that is $C_x \triangleq \left\{c \in C : \sum_{s \in S} x(c,s) = k\right\}$. The profit of a service assignment x is the total profit of satisfied clients, or $p(C_x) \triangleq \sum_{c \in C_x} p(c)$, and the goal in k-SA is to find a service assignment with maximum profit.

Given a constant $r \in (0,1]$, a k-SA instance is said to be r-restricted if no client requires more than an r-fraction of the capacity of any neighboring server, namely if $d(c) \leq r \cdot cap(s)$, for every $(c,s) \in E$. k-SA on r-restricted instances is referred to as r-restricted k-SA.

Definitions, Notation, and Assumptions. We use standard graph theoretic notation. The neighborhood of a vertex v is denoted by $N(v)$, and the degree of v is denoted by $\deg(v)$.

If a function is applied to a finite set, then this yields the sum of function values for all elements of the set, e.g., $d(C) \triangleq \sum_{c \in C} d(c)$. Also, given a function f with a finite domain, let $f_{\min}$ and $f_{\max}$ denote the minimum and maximum value of f in its domain. For example, $p_{\max} \triangleq \max_{c \in C} p(c)$.

Given a k-SA instance and a k-service assignment x, the set of clients assigned to a server s is denoted by $C_x(s) = \{c \in C : x(c,s) = 1\}$, and note that $C_x = \cup_{s \in S} C_x(s)$. We call $d(C_x(s)) = \sum_{c \in C} x(c,s)d(c)$ the *load* of server s. In this paper we sometimes consider non-feasible k-service assignments that violate the server capacity constraints, in which case it is possible that $d(C_x(s)) > cap(s)$, for a server $s \in S$. Such a server is called *overloaded*. Given $\alpha \in [0,1]$, a server s is called α-*saturated* if $d(C_x(s)) \geq \alpha \cdot cap(s)$. A service assignment x is called α-*maximal*, if no unsatisfied client is adjacent to k non-α-saturated servers.

Given a k-SA instance, the *cost effectiveness* of a client $c \in C$ is denoted by $\rho(c) \triangleq \frac{p(c)}{d(c)}$. Cost-effectiveness is assumed to be polynomially bounded, i.e., $\rho(c) \geq \rho_{\min} \in n^{-O(1)}$ as well as $\rho(c) \leq \rho_{\max} \in n^{O(1)}$. The bounds $\rho_{\min}$ and $\rho_{\max}$ are assumed to be known to each node.

Following [18] we assume that each server s is aware of the demands and profits of adjacent clients, namely each server knows $d(c)$ and $p(c)$, for every $c \in N(s)$. Observe that even if the numbers are large, it may be the case that their encoding is somewhat small (i.e., of size $O(\log n)$). An actual implementation may use a floating-point encoding, so it may be possible to efficiently send the demands and profits of clients to the adjacent servers. We also consider an alternative assumption that all nodes know the maximum profit $p_{\max}$. Notice that while the latter assumption requires global knowledge, the former requires only local knowledge.

Execution Model. We use the classic CONGEST model [19], which is a network model with small messages. Briefly, in this model nodes are processors with unique IDs, connected by links that can carry $O(\log n)$-bit messages in a time unit, or *round*. Processors are not restricted computationally (all computations required by our algorithms are polynomial, though). As usual, for our upper bounds, we implicitly assume that the α-synchronizer [2] is employed in the system, so that the algorithms operate in a synchronous manner in the following sense. Execution proceeds in global *rounds*, where in each round each processor: (i) eceives messages sent by its neighbors in the previous round, (ii) performs a local computation, and (iii) sends (possibly distinct) messages to its neighbors.

3 Centralized Greedy Algorithm

In this section we present an algorithm that computes α-maximal k-service assignments. This algorithm is used to obtain three results: (i) a $\frac{k+1-r}{1-r}$-approximation algorithm for r-restricted k-SA, for any $r \in (0,1)$, (ii) a $(k+1)$-approximation algorithm for r-restricted k-SA, for any $r \in (0,1]$, using a resource augmentation factor of $1+r$, and (iii) a $(k+1)^2$-approximation algorithm for k-SA. The first algorithm extends the $\frac{2-r}{1-r}$-approximation algorithm for 1-SA from [18]. However, we provide a simplified analysis that does not use the local ratio technique. Also, note that while the $\frac{k+1-r}{1-r}$-approximation algorithm requires knowledge of r, the other two algorithms do not.

Algorithm **α-Greedy** (Algorithm 1) sorts the clients in a non-increasing order by cost effectiveness and then tries to service the clients in order. It assigns each client to some k adjacent servers that are not yet α-saturated, if possible; otherwise, the client is dismissed. We note that if $\alpha > 1 - r$, the computed solution x may be non-feasible.

▶ **Observation 1.** *Algorithm **α-Greedy** computes α-maximal service assignments.*

Proof. Assume that the computed solution x is not α-maximal. Then, there exists a client $c_i \in C$ that is adjacent to k non-α-saturated servers. It follows that when c_i is considered by **α-Greedy** these k servers are non-α-saturated, which means that c_i would have received service. A contradiction. ◀

Algorithm 1 : $\boldsymbol{\alpha}$**-Greedy**(C, S, E, d, p)

```
Let $\langle c_1, c_2, c_3, \ldots \rangle$ be a sequence of all clients sorted in non-increasing order of $\rho$
$x \leftarrow 0$
for $i = 1, 2, 3, \ldots$ do
    if there exist $k$ non-$\alpha$-saturated servers in $N(c_i)$ then
        Let $s_1, \ldots, s_k \in N(c_i)$ be $k$ non-$\alpha$-saturated servers
        $x(c_i, s_j) \leftarrow 1$, for every $j$
    end if
end for
```

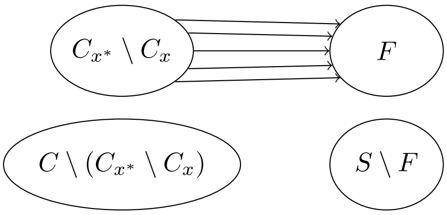


Figure 1 The arrows represent the mapping f.

We will in later sections be using rounding and therefore state our analysis of $\boldsymbol{\alpha}$**-Greedy** more generally than will be used in this section.

Let $\pi, \delta \geq 1$. Given a k-SA instance, let p' be a profit vector such that $p'(c) \in [p(c), \pi \cdot p(c)]$ and let d' be a demand vector such that $d'(c) \in [d(c), \delta \cdot d(c)]$. Define $\rho'(c) \triangleq \frac{p'(c)}{d'(c)}$.

▶ **Lemma 2.** *Given a k-SA instance, let x be the solution computed by* $\boldsymbol{\alpha}$***-Greedy*** *using p' and d', and let x^* be an optimal solution with respect p and d. Then, we have that $p(C_x) \geq \frac{\alpha}{\delta\pi k + \alpha} p(C_{x^*})$.*

Proof. Let F be the set of servers that are α-saturated with respect to the $\boldsymbol{\alpha}$**-Greedy** solution x. Consider a client $c_i \in C_{x^*} \setminus C_x$ satisfied by the optimal solution x^* but not by $\boldsymbol{\alpha}$**-Greedy**. Since $\boldsymbol{\alpha}$**-Greedy** does not satisfy c_i and due to Observation 1, c_i must be connected to fewer than k non-α-saturated servers (in $S \setminus F$). Therefore, there exists an α-saturated server $s \in F$ that is assigned to c_i by the optimal solution, that is such that $x^*(c_i, s) = 1$. Let f be a mapping which maps each client $c_i \in C_{x^*} \setminus C_x$ to a server $s \in F$ such that $x^*(c_i, s) = 1$. This is depicted in Figure 1.

Observe that the load of an α-saturated server s is by definition at least

$$d'(C_x(s)) = \sum_c x(c, s) d'(c) \geq \alpha \cdot cap(s) \ .$$

Let $f^{-1}(s) = \{c \in C_{x^*} \setminus C_x : f(c) = s\}$ be the set of clients that are mapped to a α-saturated server $s \in F$. Since each such client is assigned to s in the optimal solution x^*,

$$d(f^{-1}(s)) \leq \sum_c x^*(c, s) d(c) \leq cap(s) \leq \frac{d'(C_x(s))}{\alpha} \ . \tag{1}$$

Consider a client $c_i \in f^{-1}(s)$ and a client $c_j \in C_x(s)$. Since x does not satisfy c_i, the server s must have been α-saturated when $\boldsymbol{\alpha}$**-Greedy** tried to assign c_i. Thus, c_j must have been considered by $\boldsymbol{\alpha}$**-Greedy** prior to c_i, and the cost-effectiveness of c_j is then at least as high as that of c_i, i.e., $\rho'(c_j) \geq \rho'(c_i)$. It follows that $\rho'(c) \leq \rho'(c')$, for every $c \in f^{-1}(s)$ and $c' \in C_x(s)$. This implies that $\rho'(c) \leq \frac{p'(C_x(s))}{d'(C_x(s))}$, for every $c \in f^{-1}(s)$.

For the total profit of all clients that f maps to s we then have that

$$\begin{aligned} p(f^{-1}(s)) \;\le\; p'(f^{-1}(s)) &= \sum_{c\in f^{-1}(s)} d'(c)\cdot\rho'(c) \\ &\le \sum_{c\in f^{-1}(s)} d'(c)\cdot\frac{p'(C_x(s))}{d'(C_x(s))} \\ &= d'(f^{-1}(s))\cdot\frac{p'(C_x(s))}{d'(C_x(s))} \\ &\le \delta\cdot d(f^{-1}(s))\cdot\frac{p'(C_x(s))}{d'(C_x(s))} \;\le\; \frac{\delta}{\alpha}p'(C_x(s)) \;\le\; \frac{\pi\delta}{\alpha}p(C_x(s))\ , \end{aligned}$$

where the third inequality is due to (1).

It remains to bound the approximation ratio:

$$\begin{aligned} p(C_{x^*}) &= \sum_{c\in C_{x^*}\cap C_x} p(c) \;+\; \sum_{c\in C_{x^*}\setminus C_x} p(c) \\ &\le \sum_{c\in C_x} p(c) \;+\; \sum_{s\in F} p(f^{-1}(s)) \\ &\le p(C_x) \;+\; \sum_{s\in F}\frac{\pi\delta}{\alpha}p(C_x(s)) \;\le\; p(C_x) \;+\; k\cdot\frac{\pi\delta}{\alpha}p(C_x) \;=\; \frac{\alpha+\pi\delta k}{\alpha}\cdot p(C_x)\ , \end{aligned}$$

where the last inequality holds because each client is assigned to k servers. ◀

We note that a similar proof can be given with comparison to an optimal fractional solution as was done in [1] for the case of $k = 1$.

Furthermore, we show that the analysis of **α-Greedy** is almost tight. Consider the following k-SA instance for the case where $1-\alpha = \frac{1}{q}$, for $q \in \mathbb{N}$. Let $C = \{c_1, c_2, \ldots\}$ be a set of $q(k+1)-1$ clients and $S = \{s_1, s_2, \ldots\}$ be a set of $2k-1$ servers. For $i \le q-1$, let $d(c_i) = q$, $p(c_i) = q^t + 1$, and $N(c_i) = \{s_1, \ldots, s_k\}$, while for $i \ge q$, let $d(c_i) = q$, $p(c_i) = q^t$, and $N(c_i) = S$. As for server capacities, $cap(s_i) = q^2$, for $i \le k$, and $cap(s_i) = kq^2$, for $i > k$. If we run **α-Greedy** (assuming $\delta = \pi = 1$), it will consider clients $c_1, \ldots, c_{q-1}$ first and assigns all of them. This renders servers $s_1, \ldots, s_k$ α-saturated, so that no other clients will receive service. Thus, **α-Greedy** obtains a profit of $(q-1)(q^t+1)$, while an optimal solution services clients $c_q, \ldots, c_{q(k+1)-1}$ for a profit of $kq \cdot q^t = kq^{t+1}$. Hence, the approximation ratio of **α-Greedy** is at least $\frac{kq^{t+1}}{(q-1)(q^t+1)}$, which goes to $\frac{k}{\alpha}$ as t goes to infinity.

We get our first result by assigning $\alpha = 1-r$ and $\delta = \pi = 1$.

▶ **Corollary 3.** *If $\delta = \pi = 1$, the approximation ratio of $(1-r)$-**Greedy** is at most $\frac{k+1-r}{1-r}$.*

Our next result is obtained by assigning $\alpha = 1$ and $\delta = \pi = 1$. Notice that in this case the server capacity constraints may be violated, but not by much.

▶ **Lemma 4.** *Given an r-restricted k-SA instance, let x be a k-service assignment computed by 1-**Greedy** with $\delta = \pi = 1$. Then, the load on any server s is less than $(1+r)\cdot cap(s)$. Moreover, if we remove the last client assigned to each overloaded server we obtain a feasible k-service assignment.*

Proof. By the algorithm design, an overloaded server s was non-1-saturated when the last client was assigned to it. The load of a non-1-saturated server is less than its capacity, while the last client assigned to s has a demand of at most $r\cdot cap(s)$. ◀

From Lemma 4 we get that 1-**Greedy** obtains an approximation ratio of $k+1$ with a resource augmentation factor $(1+r)$.

▶ **Corollary 5.** *If $\delta = \pi = 1$, then* 1-***Greedy*** *is a $(k+1)$-approximation algorithm for r-restricted k-SA that uses $(1+r)$ times the capacity of each server.*

In the next lemma we show that the non-feasible solution that is computed by 1-**Greedy** can be partitioned into $k+1$ feasible solutions.

▶ **Lemma 6.** *Given a k-SA instance, let x be a k-service assignment computed by* 1-***Greedy*** *with $\delta = \pi = 1$. Then, x can be partitioned into $k+1$ feasible k-service assignments.*

Proof. Consider the directed conflict graph $G' = (C_x, E')$, where E' contains an arc (c, c') if and only if 1-**Greedy** assigned both c and c' to a server s and c was the last client assigned to s. The maximum in-degree of G' is at most k for the simple reason that x assigns at most k servers to each client. Furthermore, the graph G' is a DAG, since an edge (c, c') always points from a client c to a client c' that was considered by 1-**Greedy** prior to c. It follows that the underlying graph of G' is k-*degenerate* (or k-*inductive*), and therefore can be $(k+1)$-colored [10]. For completeness, we provide the following simple recursive algorithm that $(k+1)$-colors G':

- If the graph is empty, return an empty coloring.
- Find a node v with out-degree zero. Such a node always exists as the graph is a DAG.
- Remove v from the graph and color the remaining graph recursively.
- Color node v with the smallest available color. Since only k neighbors of v have already received a color – only the in-neighbors – at least one of the first $k+1$ colors is free.
- Return the coloring.

The coloring of G' is a partition of C_x into $k+1$ independent sets. We show that an independent set induces a feasible solution. Let I be an independent set and let x^I be the solution restricted to I, that is $x^I(c, s) = x(c, s)$, if $c \in I$, and $x^I(c, s) = 0$, otherwise. If I contains the last client assigned to a server s, then I does not contain any other clients assigned to s. It follows that x^I is feasible. ◀

This leads to the last result of the section.

▶ **Corollary 7.** *There exists a $(k+1)^2$-approximation algorithm for k-SA.*

Proof. First, 1-**Greedy**, with $\delta = \pi = 1$, computes a possibly non-feasible $(k+1)$-approximate service assignment due to Corollary 5. Lemma 6 implies that x can be partitioned into $k+1$ feasible solutions $x^1, \ldots, x^{k+1}$. Since $p(C_x) = \sum_{i=1}^{k+1} p(C_{x^i})$, there exists i such that $p(C_{x^i}) \geq \frac{1}{k+1} p(C_x)$. ◀

4 Distributed Greedy Algorithm

In this section, we present distributed approximation algorithms for k-SA by providing a distributed implementation of Algorithm $\boldsymbol{\alpha}$-**Greedy**. More specifically, we present (i) a $\frac{k+1-r}{1-r}(1+\gamma)$-approximation algorithm for r-restricted k-SA, (ii) a $(k+1)(1+\gamma)$-approximation algorithm that uses a resource augmentation factor $1+r$ for r-restricted k-SA, and (iii) a $O(k^2)$-approximation algorithm for k-SA, all for any constant $\gamma > 0$. The three algorithms terminate in $O(k^2\gamma^{-2} \operatorname{polylog}(n))$ rounds.

We first give a distributed algorithm that relies on the assumption that all nodes know $p_{\max}$. We then give a modification that does not need this assumption, but relies on the assumption that each server s knows the demands and profits of the clients in $N(s)$. We start the section by classifying the clients.

4.1 Client Classification

The basic idea of our distributed algorithm is to mimic the sequential **α-Greedy**. The challenge is to parallelize the computation of the assignment as dealing with clients one-by-one would yield linear running time. The key is to efficiently compute the assignment of multiple clients with equal profit and equal demand. To enlarge the number of clients with equal profit and demand, we apply an implication of Lemma 2: we may round profits and demands up to the closest power of $1+\varepsilon$, for some $\varepsilon > 0$, increasing the approximation ratio by at most a factor of $(1+\varepsilon)^2$.

We first classify all clients by demand and profit. Define

$$C_\ell^i \triangleq \left\{c \in C : d(c) \in ((1+\varepsilon)^{i-1}, (1+\varepsilon)^i] \ \wedge \ p(c) \in ((1+\varepsilon)^{i+\ell-1}, (1+\varepsilon)^{i+\ell}]\right\} ,$$

and $C_\ell \triangleq \bigcup_i C_\ell^i$. Also, define the *rounded* demand and profit for all clients $c \in C_\ell^i$ as

$$\begin{aligned} d'(c) &\triangleq (1+\varepsilon)^i \\ p'(c) &\triangleq (1+\varepsilon)^{i+\ell} . \end{aligned}$$

Note that all clients in C_ℓ have equal cost-effectiveness with respect to the rounded profits and demands, namely $\rho'(c) = \frac{p'(c)}{d'(c)} = (1+\varepsilon)^\ell$. That means that the clients in C_ℓ can be considered by Algorithm **α-Greedy** in any order. Also note that

$$\begin{aligned} d'(c) &\in [d(c), (1+\varepsilon)d(c)) \\ p'(c) &\in [p(c), (1+\varepsilon)p(c)) \\ \rho'(c) &\in ((1+\varepsilon)^{-1}\rho(c), (1+\varepsilon)\rho(c)) . \end{aligned}$$

For the remainder of the section, we mostly consider rounded profits and demands.

4.2 Distributed Implementation of α-Greedy

We are ready to describe a distributed implementation of Algorithm **α-Greedy** that relies on the assumption that all nodes know $p_{\max}$. The algorithm is described in a top-down manner.

By assumption, the cost-effectiveness is polynomially bounded in n. Given the values $\rho_{\min}$ and $\rho_{\max}$, we can find an interval $[W, W']$ such that $C_\ell \neq \emptyset$ only if $\ell \in [W, W']$. This is the case for $W = \lfloor \log_{1+\varepsilon}(\rho_{\min}) \rfloor$ and $W' = \lceil \log_{1+\varepsilon}(\rho_{\max}) \rceil$. Note that $W' - W \in O(\log_{1+\varepsilon} n) \subseteq O(\varepsilon^{-1} \log n)$.

Define $C_{\geq z} \triangleq \bigcup_{\ell \geq z} C_\ell$ and assume that there is an algorithm called **Augment** that augments a k-service assignment for $C_{\geq \ell+1}$ into a k-service assignment for $C_{\geq \ell}$. Algorithm **Dist-α-Greedy** (Algorithm 2) uses **Augment** iteratively to construct a k-service assignment. Clearly **Dist-α-Greedy** runs for $O(T_A \varepsilon^{-1} \log_n)$ rounds, where T_A is the running time of **Augment**.

Algorithm 2 : Dist-α-Greedy(C, S, E, d, p, cap)

```
$x \leftarrow 0$
for $\ell = W'$ downto $W$ do
    $x \leftarrow$ Augment$(C, S, E, cap, \ell, x)$
end for
```

As shown in the sequel, Algorithm **Augment** considers the subclasses of C_ℓ one by one and augments the given k-service assignment with a k-service assignment for each C_ℓ^i. In order to keep the running time of our algorithm poly-logarithmic, we use the next result showing that only considering $O(\log_{1+\varepsilon} n)$ subclasses per class C_ℓ does not increase the approximation ratio by much.

A client $c \in C$ is called *heavy* if $p'(c) > \frac{p'_{\max}}{n^3}$. Otherwise, it is called *light*. Recall that $p'_{\max} = \max_{c \in C} p'(c)$. Define $C_{\text{heavy}} \triangleq \{c \in C : p'(c) > \frac{p'_{\max}}{n^3}\}$. The next lemma explains why we can simply ignore light clients.

▶ **Lemma 8.** *Let x be an optimal k-service assignment and let y be an optimal k-service assignment for the same instance but restricted to a set $\tilde{C}$, where $\tilde{C} \supseteq C_{heavy}$. Then $p(C_x) \leq (1 + \frac{1}{n^2})p(C_y)$.*

Proof. Observe that each client $c \notin \tilde{C}$ satisfies $p(c) \leq \frac{p_{\max}}{n^3}$. Clearly, $p(C_x) \geq p(C_y) \geq p_{\max}$, and thus we have that

$$p(C_x) = p(C_x \cap \tilde{C}) + p(C_x \setminus \tilde{C}) \leq p(C_y) + n \cdot \frac{p_{\max}}{n^3} = p(C_y) + \frac{p_{\max}}{n^2} \leq \left(1 + \frac{1}{n^2}\right) p(C_y)$$

which concludes the proof. ◀

Following the above result, Algorithm **Augment** (Algorithm 3) considers only the subclasses C_ℓ^i which contain heavy clients. The heavy clients are contained in at most $\lceil 3 \log_{1+\varepsilon} n \rceil$ subclasses of C_ℓ. For each subclass, **Augment** uses Algorithm **Uniform-Augment** which augments the current k-service assignment with an assignment for the specified subclass C_ℓ^i. Recall that all clients in C_ℓ^i have the same profit and demand (with respect to p' and d').

Algorithm 3 : **Augment**(C, S, E, cap, ℓ, x)

```
$i_\ell^{\max} \leftarrow \log_{1+\varepsilon} p'_{\max} - \ell$
for $i = i_\ell^{\max}$ downto $i_\ell^{\max} - \lceil 3 \log_{1+\varepsilon} n \rceil + 1$ do
    $x \leftarrow$ Uniform-Augment$(C, S, E, cap, i, \ell, x)$
end for
return $x$
```

Clearly, if Algorithm **Uniform-Augment** requires T_U rounds, then **Augment** terminates after $O(T_U \log_{1+\varepsilon} n) \subseteq O(T_U \varepsilon^{-1} \log n)$ rounds. It follows that Algorithm **Dist-α-Greedy** requires $O(T_U \varepsilon^{-2} \log^2 n)$ rounds.

As mentioned before, Algorithm **Uniform-Augment** is used to compute a k-service assignment for all clients in a given subclass C_ℓ^i that augments a given k-service assignment x. Recall that clients have *uniform* demands, i.e., $d'(c) = (1+\varepsilon)^i$ for each $c \in C_\ell^i$. Hence, given a solution x and a server s, an upper bound $m_\ell^i(x, s)$ on the number of clients from C_ℓ^i that can be assigned to s while it is not α-saturated can be computed as follows:

$$m_\ell^i(x, s) = \min\left\{\max\left\{0, \left\lceil \frac{\alpha \cdot cap(s) - d'(C_x(s))}{(1+\varepsilon)^i} \right\rceil\right\}, \deg(s)\right\} .$$

A *star* centered at a client $c \in C$ is a subgraph of G that contains c and k servers adjacent to c. We call the servers the leaves of the star. Per server $s \in S$ we introduce $m_\ell^i(x, s)$ copies denoted s_1, s_2, and so forth. An *incarnation* of a star replaces each leaf s with a copy s_q, where $1 \leq q \leq m_\ell^i(x, s)$. Note that incarnations never have two leaves which are copies of the same server. Also note that some stars have no incarnations, namely if $m_\ell^i(x, s) = 0$ for some leaf s. We define the graph $H(i, \ell, x)$. The vertex set of $H(i, \ell, x)$

contains all possible incarnations of stars centered at a client $c \in C_\ell^i$. There is an edge between two nodes of $H(i,\ell,x)$, namely between two incarnations, if and only if the two are either centered at the same client or share a common leaf (copy of a server). Given i, ℓ, and x, Algorithm **Uniform-Augment** (Algorithm 4) constructs $H(i,\ell,x)$ and computes a *maximal independent set* (MIS) in $H(i,\ell,x)$.

Algorithm 4 : Uniform-Augment$(C, S, E, cap, i, \ell, x)$

```
$MIS \leftarrow$ Maximal Independent Set of $H(i,\ell,x)$
Augment $x$ with $k$-service assignment corresponding to $MIS$
return $x$
```

The computation of the MIS is based on Luby's algorithm [16]. In fact we rely on the analysis of Wattenhofer [24] that shows that the MIS algorithm terminates with high probability after $O(\log N)$ rounds, where N is the number of nodes in the graph. The next lemma shows how to implement the algorithm such that the number of rounds is $O(k^2 \log n)$.

▶ **Lemma 9.** *Algorithm **Uniform-Augment** computes an α-maximal service assignment w.h.p. in $O(k^2 \log n)$ rounds.*

Proof. Consider the graph $H(i,\ell,x) = (V(i,\ell,x), E(i,\ell,x))$. For a client $c \in C$, there are $\binom{\deg(c)}{k}$ stars centered at c. For each star there are at most n^k different incarnations, as there are at most $\deg(s) \le n$ copies of each server s. It follows that, per client c of G, the vertex set $V(i,\ell,x)$ contains at most $\binom{\deg(c)}{k} n^k \le n^{2k}$ vertices. So in total, $V(i,\ell,x)$ contains

$$n(i,\ell,x) \triangleq |V(i,\ell,x)| \le n^{2k+1}$$

vertices (incarnations of stars of G).

We would like to execute Luby's algorithm [16] to compute a maximal independent set in $H(i,\ell,x)$. Let $M = \emptyset$ and $\overline{M} = V(i,\ell,x)$. Luby's algorithm repeatedly executes the following procedure:

- Each incarnation in $\overline{M}$ is assigned a random priority with $O(k \log n)$ bits.
- Let U be the set of incarnations with a priority higher than any adjacent incarnation.
- All incarnations in U are added to M and removed from $\overline{M}$. Also, all incarnations adjacent to incarnations in U are removed from $\overline{M}$.

With high probability, M is a maximal independent set after the above procedure has been executed $O(k \log n)$ times [24].

We now describe how the above procedure can be simulated on the graph G:

1. A client $c \in C_\ell^i$ draws a random priority for each incarnation in $\overline{M}$ centered at c. Per client, only the incarnation with the highest priority is relevant to Luby's algorithm. Thus for each leaf s_q of such an incarnation, clients send the priority and the index q to s. Note that each client sends at most one message to each adjacent server.
2. Per copy, each server determines the highest priority received. The server sends an ACK message to the clients that sent the winning (highest) priorities and a NACK message to clients that sent the losing priorities.
3. If a client c receives k ACK messages, then the incarnation with the highest priority centered at c joins the independent set and all other incarnations centered at c are removed from $\overline{M}$. Per leaf s_q of an incarnation joining the MIS, the clients inform server s that the copy with index q has been taken.
4. Servers keep track which copies have been taken and inform the clients which copies are no longer available. Clients remove all incarnations with unavailable leaves from $\overline{M}$.

We examine the messages exchanged during this procedure and their sizes in bits. In the first step each client sends a priority and server copy index to k servers, therefore the message size is $O(k \log n + \log n) \subseteq O(k \log n)$. In the second step each server sends a single ACK/NACK message to clients whose size is $O(1)$. Winning clients send a single server copy index to k servers. The message size is $O(\log n)$. Finally, servers need to update clients on which copy cannot be used anymore. The naive solution is a message that may require $\Omega(n)$ bits (a bit vector with one bit per server copy).

Recall that for each star there are $O(n^k)$ different incarnations. Since these incarnations are interchangeable, server copies may be relabeled after each iteration such that the available copies have the smallest possible indexes. It follows that the number of available copies per server, and not the actual server copy indexes, is important. In conclusion, it suffices to inform the clients only about the number of available copies per server. This can be done using a message of size $O(\log n)$ bits.

The above procedure takes $O(1)$ rounds when assuming messages of size $O(k \log n)$ or $O(k)$ rounds using messages of size $O(\log n)$ bits. As the procedure needs to be executed $O(k \log n)$ times, we have that the total number of rounds is $O(k^2 \log n)$.

Finally, the computed solution is α-maximal, since otherwise the independent set in $H(i, \ell, x)$ is not maximal. ◀

We note that a client c need not draw one random priority for each incarnation in $\overline{M}$ centered at c. As c is aware of the number of available copies per adjacent server, it is easy to count the number of incarnations in $\overline{M}$ centered at c. Let z be this number. It then suffices to choose one of the z incarnations uniformly at random and to draw its priority from the distribution of the maximum over z random priorities (see, e.g., [9]).

We bound the running time of Algorithm **Dist-α-Greedy**.

▶ **Lemma 10.** *Algorithm **Dist-α-Greedy** terminates w.h.p. in $O(k^2\varepsilon^{-2} \cdot \log^3 n)$ rounds.*

Proof. Algorithm **Dist-α-Greedy** consists of $O(\varepsilon^{-1} \log n)$ invocations of **Augment**, which in turn consists of $O(\varepsilon^{-1} \log n)$ invocations of **Uniform-Augment**. The lemma follows since **Uniform-Augment** requires $O(k^2 \log n)$ rounds according to Lemma 9. ◀

Next, we analyze the computed solution. In preparation for Section 4.3, the next result is slightly more general than necessary.

▶ **Lemma 11.** *Given a k-SA instance, Algorithm **Dist-α-Greedy** mimics **α-Greedy** on a set $\tilde{C} \supseteq C_{heavy}$ using the rounded profits p' and the rounded demands d'.*

Proof. Notice that Algorithm **Dist-α-Greedy** considers $C_{W'}, \ldots, C_W$ in decreasing order of ℓ. Since all clients in C_ℓ have the same cost-effectiveness, $(1+\varepsilon)^\ell$, it follows that the algorithm augments x according to a non-increasing order of client cost-effectiveness with respect to p' and d'. For each ℓ, Algorithm **Augment** considers subclasses C_ℓ^i in a decreasing order of i, that is in a decreasing order of both profit and demand. For each i, **Uniform-Augment** computes an α-maximal solution, as shown in Lemma 9, by adding clients with the same profit, demand, and cost-effectiveness in an order that is induced by the random choices of the maximal independent set computation. Hence, Algorithm **Augment** can be seen as trying to service clients with the same cost-effectiveness in an arbitrary order. It follows that **Dist-α-Greedy** is a specific implementation of **α-Greedy**.

It remains to show that **Augment** considers all heavy clients in each class C_ℓ. Let $c \in C_\ell$ be a client not considered by **Augment**. Then $c \in C_\ell^i$ with $i \leq i_\ell^{\max} - \lceil 3 \log_{1+\varepsilon} n \rceil$ and we have that

$$p'(c) = (1+\varepsilon)^{i+\ell} \le (1+\varepsilon)^{i_\ell^{\max}+\ell-\lceil 3\log_{1+\varepsilon} n\rceil} = (1+\varepsilon)^{\log_{1+\varepsilon} p'_{\max}-\lceil 3\log_{1+\varepsilon} n\rceil} \le \frac{p'_{\max}}{n^3} ,$$

and thus c is a light client. ◀

The previous lemma allows us to find a lower bound on the profit of the solution that is computed by **Dist-α-Greedy**.

▶ **Lemma 12.** *Given a k-SA instance, let x be the solution computed by **Dist-α-Greedy** using p' and d', and let x^* be an optimal solution with respect p and d. Then, we have that $p(C_x) \ge \frac{1}{1+1/n^2} \cdot \frac{1}{(1+\varepsilon)^2} \cdot \frac{\alpha}{k+\alpha} p(C_{x^*})$.*

Proof. Let y^* be an optimal solution with respect to p, d, and $\tilde{C} \supseteq C_{\text{heavy}}$. We have that $p(C_{x*}) \le (1+\frac{1}{n^2})p(C_{y^*})$ by Lemma 8. Furthermore, $p(C_x) \ge \frac{1}{(1+\varepsilon)^2} \cdot \frac{\alpha}{k+\alpha} p(C_{y^*})$ due to Lemmas 2 and 11 and the definition of p' and d'. The claim follows. ◀

▶ **Lemma 13.** *Let $\gamma > 0$ be a constant. There exists distributed $((1+\gamma)\frac{k+\alpha}{\alpha})$-approximation algorithm for k-SA that terminates w.h.p. in $O(k^2\gamma^{-2} \cdot \log^3 n)$ rounds.*

Proof. If $\gamma < \frac{4}{n^2}$, then $n \le 2/\sqrt{\gamma}$ which means that $n = O(1)$. In this case, an optimal solution can be computed in $O(1)$ rounds as follows: each node sends its input to the node with highest id, which computes an optimal solution and broadcasts it to all nodes.

If $\gamma \ge \frac{4}{n^2}$, then set $\varepsilon = \gamma/4$ and run **Dist-α-Greedy**. In this case we have that

$$(1+\frac{1}{n^2}) \cdot (1+\varepsilon)^2 \le \left(1+\frac{\gamma}{4}\right)^3 = \left(1+\frac{3\gamma}{4}+\frac{3\gamma^2}{16}+\frac{\gamma^3}{64}\right) < 1+\gamma .$$

The rest follows from Lemmas 10 and 12. ◀

By setting $\alpha = 1-r$, Lemma 13 leads to the following result:

▶ **Corollary 14.** *There exists a distributed $((1+\gamma)\frac{k+1-r}{1-r})$-approximation algorithm for r-restricted k-SA that terminates w.h.p. in $O(k^2\gamma^{-2} \cdot \log^3 n)$ rounds, for every $\gamma > 0$.*

We can obtain a better ratio using resource augmentation, i.e., by setting $\alpha = 1$.

▶ **Corollary 15.** *There exists a distributed $(1+\gamma)(k+1)$-approximation algorithm for r-restricted k-SA that uses at most $(1+r)$ times the capacity of each server and terminates w.h.p. in $O(k^2\gamma^{-2} \cdot \log^3 n)$ rounds, for every $\gamma > 0$.*

As in the centralized case (Lemma 6) we use the resource augmentation algorithm in order to obtain a feasible service assignment. However, in the distributed setting we use random selection instead of using coloring.

▶ **Theorem 16.** *There exists a distributed algorithm for k-SA that terminates w.h.p. in $O(k^2\gamma^{-2} \cdot \log^3 n)$ rounds and computes solutions whose expected profit is at least $p(C_{x^*})/((1+\gamma) \cdot 4k(k+1))$, for any $\gamma > 0$, where x^* is an optimal solution.*

Proof. We present a distributed randomized algorithm that computes a service assignment whose expected profit is at least the optimum divided by $(1+\gamma) \cdot 4k(k+1)$, for any constant $\gamma > 0$.

The first phase of the algorithm is to compute a $(1+\gamma)(k+1)$-approximate solution x for k-SA that uses at most $(1+r)$ times the capacity of each server. By Corollary 15 this

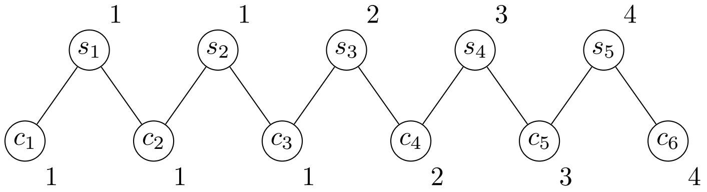


Figure 2 Each client is labeled with the index of its class, each server is labeled with $i_\ell^{\max}(s)$.

takes $O(k^2\gamma^{-2} \cdot \log^3 n)$ rounds. The solution is either already feasible or was computed by **Dist-1-Greedy** (see Lemma 13). In the latter case, consider the set of clients that were last assigned to s by an invocation of **Uniform-Augment** and choose as $c(s)$ the client with the largest identifier.

Recall the definition of the conflict graph $G' = (C_x, E')$ from Lemma 6. The set E' contains an edge between two clients (c, c') if c and c' are both assigned to a server s and $c = c(s)$. The second phase is to compute an independent set I of G'. As shown in the proof of Lemma 6, restricting x to the clients in I yields a feasible solution.

We exploit that G' is a DAG with in-degree at most k. Let $\beta > 1$ and $U = \emptyset$. Add each client $c \in C_x$ to U independently with probability $\frac{1}{\beta k}$. Then let $I \subseteq U$ be the set of clients with no in-neighbor in U. Clearly I is an independent set of G'. By the Union Bound, a node has an in-neighbor in U with probability at most $k \cdot \frac{1}{\beta k} = \frac{1}{\beta}$. Thus, a node of C_x is in I with probability at least $\frac{1}{\beta k}(1 - \frac{1}{\beta}) = \frac{\beta-1}{\beta^2 k}$. We choose $\beta = 2$ to maximize $\frac{\beta-1}{\beta^2}$, so a node of C_x is in I with probability $\frac{1}{4k}$. Thus $\mathbb{E}[p(I)] \geq \frac{1}{4k} p(C_x)$.

The set I can be easily constructed by the following distributed algorithm. Every client in $c \in C_x$ informs each server s with $x(c, s) = 1$ whether $c \in U$. A server s responds to a client $c \in U$ with a NACK message if $c \neq c(s)$ and $c(s) \in U$, and with an ACK message otherwise. If a client $c \in U$ receives no NACK message, then $c \in I$. Otherwise, c informs its servers that $c \notin I$. ◀

4.3 Modification

In the remainder of this section, we describe a modified version of **Augment**, called **Modified-Augment**, that does not assume knowledge of $p_{\max}$. Instead, we assume that each server s knows the demand $d(c)$ and the profit $p(c)$ of each adjacent client $c \in N(s)$, as explained in Section 2.

Without knowledge of $p_{\max}$, intuitively, we would like to start with the non-empty subclass C_ℓ^i with maximum index i. A naive approach, such as determining the maximum index i and making it known to all nodes, would require time proportional to the network diameter. Our algorithm avoids this issue by using the index

$$i_\ell^{\max}(s) = \max\{i : C_\ell^i \cap N(s) \neq \emptyset\} ,$$

for each server $s \in S$. See Figure 2 for an example. As the demands and profits are known, a server s can easily determine $i_\ell^{\max}(s)$.

Algorithm **Modified-Augment** works as follows. In each iteration of a loop starting at $i = i_\ell^{\max}(s)$ and counting downwards, each server s sends a START message to each adjacent client in class C_ℓ^i. It then runs Algorithm **Uniform-Augment** for index i. The execution of **Uniform-Augment** for index i is restricted to the graph $G(i, \ell) = (C_\ell^i, S, E \cap (C_\ell^i \times S))$. Thus, a client $c \in C_\ell^i$ may only receive messages due to an execution of **Uniform-Augment** for index i.

A client $c \in C_\ell^i$ doesn't run Algorithm **Uniform-Augment** for index i straightaway. Instead, it waits until all adjacent servers have sent a START message. While delaying the execution of **Uniform-Augment**, incoming messages for **Uniform-Augment** are saved by c and delivered later when its execution starts.

As messages are delayed and since **Uniform-Augment** was written for the synchronous model, we use an α-synchronizer to execute **Uniform-Augment**. Also, we assume that the synchronizer counts how many synchronous rounds **Uniform-Augment** has been executed for. This serves as a means of termination detection. Let T_U be the worst-case running time of **Uniform-Augment** in synchronous rounds. After starting an execution of **Uniform-Augment** for a particular index, servers and clients wait until the synchronizer has completed T_U synchronous rounds of **Uniform-Augment**. Once this has happened, clients update the k-service assignment and servers continue with the next iteration of the loop over i.

Consider the graph shown in Figure 2. In the first round, servers s_1 and s_2 send a START message to c_1, c_2, and c_3. So servers s_1 and s_2 as well as clients c_1 and c_2 start executing **Uniform-Augment** for class C_ℓ^1. However, client c_3 will locally delay the execution of **Uniform-Augment** for class C_ℓ^1 until it received a START message from server s_3. This will only happen after s_3 and c_4 have finished executing **Uniform-Augment** for class C_ℓ^2. This in turn will be delayed until s_4 and c_5 have finished executing **Uniform-Augment** for class C_ℓ^3. We observe that the execution of **Uniform-Augment** for some class C_ℓ^i is delayed for at most $(i_\ell - i) \cdot O(T_U)$ rounds, where $i_\ell = \max_{s \in S} i_\ell^{\max}(s)$.

As we do not need to consider all subclasses of C_ℓ but mainly subclasses with heavy clients, we simply stop the execution of **Modified-Augment** after $O(T_U \varepsilon^{-1} \log n)$ rounds and take the k-service assignment computed by then. Abruptly stopping the execution may render the local view of the computed k-service assignment by clients and servers inconsistent. This can be fixed within one round by letting each client c send the value $x(c, s)$ to each adjacent server s. As the following result shows, the given time bound suffices to let **Modified-Augment** consider all heavy clients.

▶ **Lemma 17.** *With high probability, Algorithm **Modified-Augment** requires $O(k^2 \varepsilon^{-1} \log^2 n)$ rounds to run **Uniform-Augment** for all subclasses of C_ℓ that contain heavy clients.*

Due to space constraints, the listing of Algorithm **Modified-Augment** and the proof of Lemma 17 have been omitted. Algorithm **Modified-Augment** is a drop-in replacement for Algorithm **Augment** in the sense that it preserves all significant properties of **Augment**: (i) All subclasses of C_ℓ with heavy clients are considered, (ii) **Modified-Augment** has the same asymptotic runtime as Algorithm **Augment**, and (iii) the subclasses C_ℓ^i are (locally) considered in decreasing order of index i. We conclude that in particular Lemmas 10 and 11 remain true if **Augment** is replaced with **Modified-Augment** (details are omitted). As all subsequent results in Section 4.2 are mainly derived from these two lemmas, they also remain true.

Acknowledgements. We thank Boaz Patt-Shamir for helpful discussions.

References

1 David Amzallag, Reuven Bar-Yehuda, Danny Raz, and Gabriel Scalosub. Cell selection in 4G cellular networks. *IEEE Trans. Mobile Comput.*, 12(7):1443–1455, 2013.

2 Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

3 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baurch Schieber. A unified approach to approximating resource allocation and schedualing. *J. ACM*, 48(5):1069–1090, 2001.

4 Reuven Bar-Yehuda, Keren Bendel, Ari Freund, and Dror Rawitz. Local ratio: A unified framework for approximation algorithms. *ACM Comput. Surv.*, 36(4):422–463, 2004.

5 Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.

6 Chandra Chekuri and Sanjeev Khanna. On multidimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, 2004.

7 Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 35(3):713–728, 2005.

8 Milind Dawande, Jayant Kalagnanam, Pinar Keskinocak, F. Sibel Salman, and R. Ravi. Approximation algorithms for the multiple knapsack problem with assignment restrictions. *Journal of Combinatorial Optimization*, 4(2):171–186, 2000.

9 Yuval Emek, Magnús M. Halldórsson, Yishay Mansour, Boaz Patt-Shamir, Jaikumar Radhakrishnan, and Dror Rawitz. Online set packing. *SIAM J. Comput.*, 41(4):728–746, 2012.

10 Paul Erdös and András Hajnal. On chromatic number of graphs and set-systems. *Acta Mathematica Hungarica*, 17(1–2):61–99, 1966.

11 Lisa Fleischer, Michel X. Goemans, Vahab S. Mirrokni, and Maxim Sviridenko. Tight approximation algorithms for maximum general assignment problems. In *17th SODA*, pages 611–620, 2006.

12 A. M. Frieze and M. R. B. Clarke. Approximation algorithms for the m-dimensional $0-1$ knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15:100–109, 1984.

13 Magnús M. Halldórsson, Sven Köhler, Boaz Patt-Shamir, and Dror Rawitz. Distributed backup placement in networks. In *27th ACM SPAA*, pages 274–283, 2015.

14 Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006.

15 Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

16 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

17 Michael J. Magazine and Maw-Sheng Chern. A note on approximation schemes for multidimensional knapsack problems. *Mathematics of Operations Research*, 9(2):244–247, 1984.

18 Boaz Patt-Shamir, Dror Rawitz, and Gabriel Scalosub. Distributed approximation of cellular coverage. *J. Parallel Distrib. Comput.*, 72(3):402–408, 2012.

19 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.

20 Prabhakar Raghavan and Clark D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

21 Sartaj Sahni. Approximate algorithms for the 0/1 knapsack problem. *J. ACM*, 22(1):115–124, 1975.

22 David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

23 Aravind Srinivasan. Improved approximation guarantees for packing and covering integer programs. *SIAM J. Comput.*, 29(2):648–670, 1999.

24 Roger Wattenhofer. Principles of distributed computing: Maximal independent set. `http://www.dcg.ethz.ch/lectures/fs15/podc/lecture/chapter7.pdf`. Accessed 2015-08-27.

On the Uncontended Complexity of Anonymous Consensus*

Claire Capdevielle[1], Colette Johnen[2], Petr Kuznetsov[3], and Alessia Milani[4]

1 **Univ. Bordeaux, LaBRI, UMR 5800, Talence, France**
claire.capdevielle@labri.fr
2 **Univ. Bordeaux, LaBRI, UMR 5800, Talence, France**
johnen@labri.fr
3 **Télécom ParisTech, Paris, France**
petr.kuznetsov@telecom-paristech.fr
4 **Univ. Bordeaux, LaBRI, UMR 5800, Talence, France**
milani@labri.fr

Abstract

Consensus is one of the central distributed abstractions. By enabling a collection of processes to agree on one of the values they propose, consensus can be used to implement any generic replicated service in a consistent and fault-tolerant way.

In this paper, we study *uncontended* complexity of anonymous consensus algorithms, counting the number of memory locations used and the number of memory updates performed in operations that encounter no contention. We assume that contention-free operations on a consensus object perform "fast" reads and writes, and resort to more expensive synchronization primitives, such as CAS, only when contention is detected. We call such concurrent implementations *interval-solo-fast* and derive one of the first nontrivial tight bounds on space complexity of anonymous interval-solo-fast consensus.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases space and time complexity, lower bounds, consensus, interval contention, solo-fast

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.12

1 Introduction

Consensus is one of the central distributed abstractions. By enabling a collection of processes to agree on one of the values they propose, consensus can be used to implement any generic replicated service in a consistent and fault-tolerant way. Therefore, complexity of consensus implementations has become one of the most important topics in the theory of distributed computing.

It is known that consensus cannot be solved in an asynchronous read-write shared memory system in a deterministic and fault-tolerant way [7, 16]. The difficulty stems from handling contended executions. One way to circumvent this impossibility is to only guarantee progress

* Partially supported by the ANR project DISPLEXITY (ANR-11-BS02-014). This study has been carried out in the frame of the Investments for the future Programme IdEx Bordeaux-CPU (ANR-10-IDEX-03-02). The third author was supported by the ANR project DISCMAT, under grant agreement N ANR-14-CE35-0010-01.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 12; pp. 12:1–12:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(using reads and writes) in executions meeting certain conditions, e.g., in the absence of *contention*. Alternatively, a process is guaranteed to decide in the *wait-free* manner, but stronger (and more expensive) synchronization primitives, such as *compare-and-swap*, can be applied in the presence of contention.

We are interested in consensus algorithms in which a *propose* operation is allowed to apply primitives other than reads and writes on the base objects only in the presence of *interval contention*, i.e., when another *propose* operation is concurrently active. These algorithms are called *interval-solo-fast*.

Ideally, interval-solo-fast algorithms should have an optimized behavior in *uncontended* executions. It appears therefore natural to explore the uncontended complexity of consensus algorithms: how many memory operations (reads and writes) need to be performed and how many distinct memory locations need to be accessed in the absence of interval contention?

In general, interval-solo-fast consensus can be solved with only constant uncontended complexity [17]. We therefore restrict our study to *anonymous* consensus algorithms, i.e., algorithms not using process identifiers and, thus, programming all processes identically. Besides intellectual curiosity, practical reasons to study anonymous algorithms in the shared memory model are discussed in [10].

Our results. On the lower-bound side, we show that any anonymous interval-solo-fast consensus algorithm exhibits non-trivial uncontended complexity that depends on n, the number of processes, and m, where m is the size of the set V of input values that can be proposed. More precisely, we show that, in the worst case, a *propose* operation running *solo*, i.e., without any other process invoking *propose*, must write to $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ distinct memory locations. This metrics, which we call *solo-write complexity*, is upper-bounded by the *step complexity* of the algorithm, i.e., the worst-case number of all base-object primitives applied by an individual operation. In the special case of *input-oblivious* algorithms, where the sequence of memory locations written in a solo execution does not depend on the input value, we derive a stronger lower bound of $\Omega(\sqrt{n})$ on solo-write complexity. Our proof only requires the algorithm to ensure that operations terminate in solo executions, so the lower bounds also hold for *abortable* [2, 11] and obstruction-free [13] consensus implementations.

On the positive side, we show that our lower bound is tight. Our matching consensus algorithm is based on our novel *value-splitter* abstraction, extending the classical *splitter* mechanism [15, 18, 4], interesting in its own right. Informally, a value-splitter exports a single operation *split* that takes a value in a value set V as a parameter and returns a boolean response so that (1) if *split*(v) completes before any other *split* operation starts, then it returns *true*, and (2) all processes that obtain *true* proposed the same value.

We describe a simple transformation of a value-splitter into anonymous and interval-solo-fast consensus, using the classical splitter-based algorithm and incurring constant overhead with respect to the value-splitter complexity [17]. Then, we present two value-splitter read-write implementations that combined with the consensus algorithm provide the matching upper bound $O(\min(\sqrt{n}, \log m/\log\log m))$.

The first one is a novel anonymous and input-oblivious implementation of a value-splitter that exhibits $O(\sqrt{n})$ space and solo-write complexity.

The second one is not input-oblivious, and is a slight modification of the *weak conflict detector* proposed in [1], exhibiting $O(\log m/\log\log m)$ space and step complexity.

Our results are summarized in Table 1. It is interesting to notice that the step complexities are $O(n)$ for the first algorithm and $O(\log m/\log\log m)$ for the second one. Aspnes and Ellen [1] showed that any anonymous consensus protocol has to execute

Table 1 Space and solo-write complexity for anonymous interval-solo-fast consensus.

Input-oblivious	Not input-oblivious	
$\Omega(\sqrt{n})$	$\Omega(\min(\sqrt{n}, \frac{\log m}{\log\log m}))$	
$O(\sqrt{n})$	if $\sqrt{n} \leq \frac{\log m}{\log\log m}$, $O(\sqrt{n})$	if $\sqrt{n} \geq \frac{\log m}{\log\log m}$, $O(\frac{\log m}{\log\log m})$ [17, 1]

$\Omega(\min(n, \log m/\log\log m))$ steps in solo executions. Thus, our consensus algorithms have also asympotically optimal step complexity.

Overall, our results imply one the first nontrivial *tight* lower bound on the space complexity for consensus known so far, along with a concurrent result on the space complexity of *solo-terminating* anonymous consensus [8].[1] Our results also show that there is an inherent gap between anonymous and non-anonymous consensus algorithms: non-anonymous consensus has constant uncontended complexity [17].

Related work. The idea of optimizing concurrent algorithms for uncontended executions was suggested by Lamport in his "fast" mutual exclusion algorithm [15].

Fich et al. [6] have shown that any solo-terminating (and, as a result, obstruction-free) read-write (non-anonymous) consensus protocol must use $\Omega(\sqrt{n})$ memory locations. Gelashvili [8] proved a stronger $\Omega(n)$ lower bound for the anonymous case. Attiya et al. [2] showed that any *step-solo-fast* (where operations only apply reads and writes in the absence of interleaving steps) either use $O(\sqrt{n})$ space or incur $O(\sqrt{n})$ memory *stalls* per operation. No obstruction-free or step-solo-fast algorithm matching these lower bounds is known so far: existing algorithms typically expose $O(n)$ space complexity. These lower bounds focus on *step contention* and do not extend to uncontended executions, where no interval contention is encountered.

Our *value-splitter* abstraction is inspired by the splitter mechanism in [18, 4], originally suggested by Lamport [15]. Differently from the original splitter object, more than one process can return *true* but all these processes have the same input value. The novel input-oblivious value-splitter implementation we present is inspired by the obstruction-free leader election algorithm recently proposed by Giakkoupis et al. [9].

Bouzid et al. [3] presented an anonymous consensus algorithm with asymptotically optimal solo write and step complexity. But it relies on a failure detector (can be transformed into obstruction-free though) and requires *unbounded* space.

A preliminary version of this paper has been presented as a brief announcement [5].

Roadmap. The rest of the paper is organized as follows. We give preliminary definitions in Section 2. We present our lower bound in Section 3 and our upper bound in Section 4. We conclude the paper in Section 5.

[1] Informally, a solo-terminating algorithm ensures that every process running solo from any configuration eventually terminates.

2 Preliminaries

The model of computation

We consider a standard asynchronous shared-memory model in which $n > 1$ processes communicate by applying atomic (or linearizable [14]) *primitive* operations on shared variables, called *base objects*. We assume every base object maintains a *state* and exports a subset of the *Read*, *Write* and *Compare-And-Swap* (CAS) primitives. The primitive *Read*(R) returns the value of R, and *Write*(R, v) sets the state of R to v. The primitive *CAS*(R, e, v) checks if the state of R is e and, if so, sets the state of R to v and returns *true*; otherwise, the state remains unchanged and *false* is returned. A *register* is a base object that exports only the Read and Write primitives.

Algorithms and executions

To implement a (high-level) object from a set of base objects, processes follow an *algorithm* $\mathcal{A}$, associating each process p with an deterministic automaton $\mathcal{A}_p$. To avoid confusion between the base objects and the implemented one, we reserve the term *operation* for the object being implemented and we call *primitives* the operations on base objects. We say that an operation is *performed* on a high-level object and that a primitive is *applied* to a base object.

Each process has a local state that consists of the values stored in its local variables and a programme counter. A computation of the system proceeds in *steps* of an algorithm performed by the processes. Each step is one of the following: (1) an invocation of a high-level *operation*, (2) a primitive operation on a base object that returns a response and results in a change of a process's state, or (3) a response of a (high-level) operation. A *configuration* specifies the state of each base object and the local state of each process at one moment. In an *initial configuration*, all base objects have the initial values specified by the algorithm and all processes are in their initial states.

A process is *active* if an operation has been invoked by the process but the operation has not yet produced a matching response; otherwise the process is called *idle*. We assume that an operation can only be invoked on an idle process and only active processes take steps. A configuration is *quiescent* if every process is idle in it.

An *execution fragment* of an algorithm is a (possibly infinite) sequence $C_1, \phi_1, \ldots, C_i, \phi_i, \ldots$ of configurations alternating with steps, where each step is the application of a primitive ϕ_i to configuration C_i resulting in configuration C_{i+1}. For any finite execution fragment α ending with configuration C and any execution fragment α' starting at C, the execution $\alpha\alpha'$ is the concatenation of α and α'; in this case α' is called an *extension* of α. An *execution* is an execution fragment starting from the initial configuration C_0.

In an infinite execution, a process is *correct* if it takes an infinite number of steps or is idle from some point on. Otherwise, the process is called *crashed*.

In a *solo* execution, only one process takes steps. An operation invoked by a process in a given execution is *completed* if its invocation is followed by a matching response. An operation invoked a process p in an execution E is *uncontended* if no process other than p is active between its invocation and response steps. We also say that p executes its operation in absence of *interval contention*.

Finally, we say that an operation executes in the absence of *step contention* if all the steps of the operation are contiguous in the execution.

Consensus

The *consensus* object exports one operation *propose*(v), where v is an *input* taken from some domain V ($|V| \geq 2$). The output values must satisfy the following properties:

- *Agreement*: all output values are the same
- *Validity*: Every output value is one of the input values.

Properties of algorithms

An algorithm is *wait-free* if in every execution, each correct process completes each of its operation in a finite number of its own steps [12].

A wait-free algorithm is *interval-solo-fast* if, in absence of interval contention, a process only applies Read and Write primitives. A wait-free algorithm is *step-solo-fast* [2] if a process is allowed to apply only Reads and Writes in the absence of *step contention*, i.e.. when its steps are not interleaved with the steps of another process.

An algorithm is *input-oblivious* if a process accesses the same sequence of base objects in any solo execution of the algorithm, regardless of its input.

An algorithm $\mathcal{A}$ is *anonymous* if $\mathcal{A}_p$ does not depend on p, i.e., the algorithm programs the processes identically, regardless of their identifiers.

In this paper we are concerned with two complexity metrics: *space complexity*, i.e., the number of base objects an algorithm uses, and *solo-write complexity*, i.e., the maximal number of writes performed in a solo execution of a single operation of an algorithm, taken over all possible input values. Note that solo-write complexity is upper-bounded by the *step complexity* of the algorithm, i.e., the number of base-object accesses a single operation may perform.

3 Lower bounds for interval-solo-fast consensus

Consider any n-process anonymous implementation of interval-solo-fast consensus with a set V of input values, $|V| = m$. In this section, we show that the implementation must have an execution in which some propose operation, running solo, performs $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ writes on distinct objects. Obviously, the implementation must use $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ base objects.

We also show that in the special case when the algorithm is, additionally, *input-oblivious* the lower bounds become $\Omega(\sqrt{n})$.

Overview of the proof

By the way of contradiction, assume that there exists an interval-solo-fast anonymous consensus algorithm $\mathcal{A}$ such that at most k distinct base objects are written in any solo execution of $\mathcal{A}$ and $k < \min(\sqrt{n}, \Gamma^{-1}(m))$. Here Γ^{-1} is the inverse of the factorial function $\Gamma(m) = m!$. Recall that $\Gamma^{-1}(m) = \Theta(\log m/\log\log m)$.

We are going to establish a contradiction by showing that the algorithm has an execution in which two different values are returned. In executions we are going to iteratively construct, no process encounters interval contention and, thus, no process applies primitives other than Reads and Writes.

Let C_0 be the initial configuration of $\mathcal{A}$. For each $v \in V$, let α_v denote the execution of $\mathcal{A}$ in which a process, starting from C_0, invokes *propose*(u) and runs solo and until the operation completes. Since the algorithm is anonymous, α_v does not depend on the process identifier.

For a given $v \in V$, consider the sequence of base objects written in α_v, ordered by the times they are *first written* in α_v. There are m possible values v (and, thus, possible executions α_v), and at most $k!$ possible orders in which base objects can be written for the first time in executions α_v, $v \in V$.

Since $k < \Gamma^{-1}(m)$, we have $k! < m$ and, thus, there must be two values v and w such that the sequences of base objects written in α_v and α_w, in the order of the times they are first written, are identical. (In an input-oblivious protocol, v and w can be any two distinct values, regardless of the relation between m and k.) Let us denote this sequence of base objects by $r_1, \ldots, r_k$ and fix it for the rest of the proof.

To construct the desired execution with different returned values and establish a contradiction, we assume that half of the processes propose v and the other half propose w. In each iteration of the construction, we "wake up" a subset of the processes in each of the two halves and let them run as *clones*, i.e., run them lock-step so that they ignore the presence of each other, until they are about to write to a base object for the first time. On the way, we carefully maintain the invariant that each previously written base object is *covered* by "enough" processes in each of the two halves: a process *p covers a base object r in a given configuration C* if p is about to write to r in C. Intuitively, these "covering" write operations, once applied, ensure that one half of the processes will not be able to "notice" the presence of the other half in an extended execution. As a result, in the subsequent iteration, we can extend the execution in a way that "enough" processes in each of the two halves cannot distinguish it from a solo run.

Using the assumption $k < \sqrt{n}$, we ensure that at the end of the kth iteration, we have at least one process p_i proposing v and at least one process p_j proposing w, and both p_i and p_j believe that they run solo. Moreover, in the resulting configuration C_k, each of the k base objects $r_1, \ldots, r_k$ is covered by at least one process proposing v with the value last written to it by p_i and at least one process proposing w with the value last written to it by p_j. Therefore, we can extend C_k with a block write of the processes proposing v and then let p_i run until completion, without being able to distinguish the resulting execution from α_v. Thus, p_i must eventually return v. But then we can extend the resulting execution with a block write of the processes proposing $w$and let p_j run until completion, without being able to distinguish the current execution from α_w. Thus, p_j will have to return w, which establishes the contradiction.

Notations and definitions

We now introduce some instrumental notions and definition.

Recall that α_u denotes the complete solo execution of *propose*(u) from the initial configuration C_0. For $u = v, w$, $1 \leq i \leq k$, let $\alpha_{i,u}$ denote the longest prefix of α_u which only contains writes on base objects in $\{r_1, \ldots, r_i\}$. Let $\alpha_{0,u}$ denote the longest prefix of α_u in which no writes takes place. By the definition, $\alpha_{k,u} = \alpha_u$, and for all $0 \leq i \leq k-1$, the next event of α_u immediately after $\alpha_{i,u}$ is a write on r_{i+1}.

For $j = 1, \ldots, k$, let $x_{j,i,u}$ denote the value of r_j in the configuration right after $\alpha_{i,u}$. Recall that for $j = i+1, \ldots, k$, no write on r_j takes place in $\alpha_{i,u}$ and, thus, $x_{j,i,u}$ is the initial value of r_j.

For $i, j = 1, \ldots, k$ and $u = v, w$, let $I_{j,i,u}$ be a binary indicator that r_j is written in $\alpha_{i,u}$ *after* the last event of $\alpha_{i-1,u}$. Note that $I_{i,i,u} = 1$ for all $i = 1, \ldots, k$, and $I_{j,i,u} = 0$ for all $1 \leq i < j \leq k$.

For example, consider the solo execution of a *propose*(u) operation depicted in Figure 1. Here, $I_{1,2,u}$ is the binary indicator that r_1 is written in $\alpha_{2,u}$ *after* the last event of $\alpha_{1,u}$,

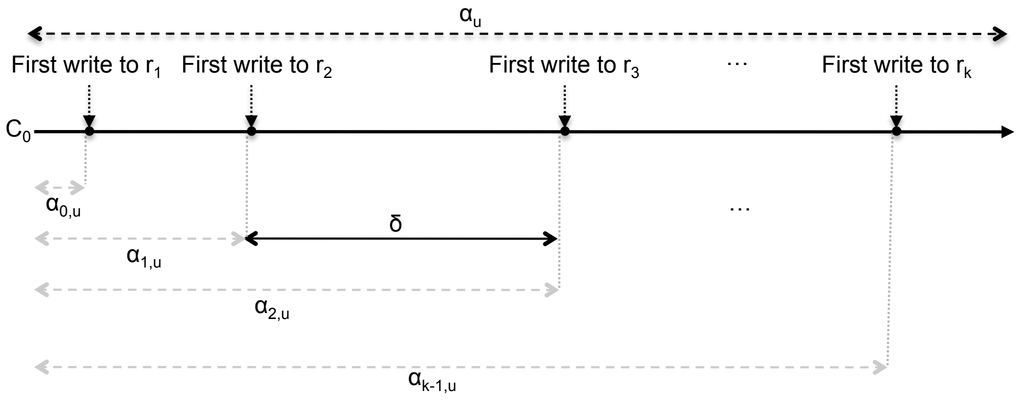


Figure 1 Solo execution of *propose*(u) by a process p, denoted α_u.

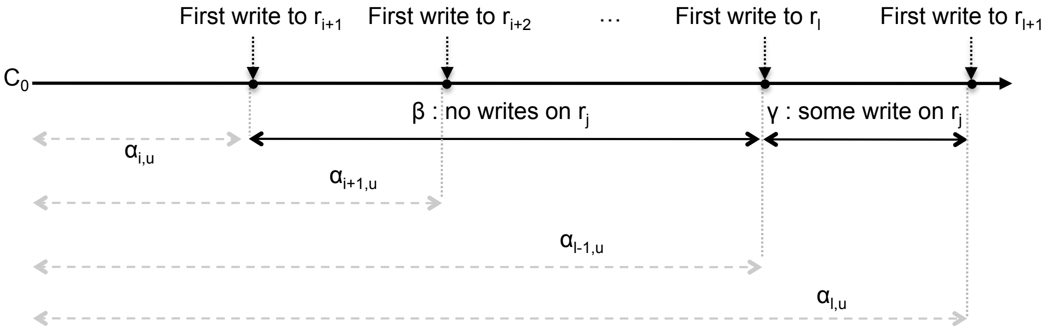


Figure 2 Definition of $s_{j,i,u}$: β contains $\ell - i$ consecutive fragments in which r_j is not written.

i.e., in the execution fragment δ. If there is a write on r_1 in δ then $I_{1,2,u} = 1$. Otherwise, $I_{1,2,u} = 0$.

For $1 \leq i, j < k$, we define $s_{j,i,u}$ as 1 plus the maximal number of *consecutive* prefixes $\alpha_{t,u}$ such that $i < t < k$ and r_j is *not* written in $\alpha_{t,u}$ after the last event of $\alpha_{t-1,u}$, i.e., $s_{j,i,u} = \min\{\ell > i | \ell = k \vee I_{j,\ell,u} = 1\} - i$.

Figure 2 depicts a fragment of the execution α_u and graphically explains the notation $s_{j,i,u}$. In particular, for a given base object r_j and a given prefix $\alpha_{i,u}$, we consider the longest sequence of consecutive distinct fragments between the first write to r_t up to (but not including) the first write to r_{t+1}, which contain no writes on r_j, starting from $t = i+1$. This sequence of fragments is denoted by β here. Then $s_{j,i,u}$ is simply the number of consecutive fragments in β plus one, i.e., $\ell - i$ in this case, as the fragment γ between the first write to r_ℓ up to the first write to $r_{\ell+1}$ contains a write to r_j.

Clearly, $s_{j,i,u} \geq 1$ and $s_{j,k-1,u} = 1$, for all $i, j = 1, \ldots, k-1$. Also, it is easy to check that $\sum_{i=1}^{k-1} I_{j,i,u} s_{j,i,u} = k - j$ for all $j = 1, \ldots, k-1$. Thus, $\sum_{\ell=1}^{k-1} \sum_{j=1}^{k-1} I_{j,\ell,u} s_{j,\ell,u} = (k^2 - k)/2$.

Cloning configurations.

We now introduce the central notion of our lower-bound proof:

▶ **Definition 1.** A configuration C_i is called *i-cloning*, $1 \leq i \leq k$, if it satisfies the following conditions:

- For each $u = v, w$, $j = 1, \ldots, i-1$, r_j is covered by $s_{j,i-1,u}$ processes writing $x_{j,i-1,u}$.

- For each $u = v, w$, there are at least $(k^2 - k + 2)/2 - \sum_{\ell=1}^{i-1} \sum_{j=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u}$ processes that do not distinguish the execution from $\alpha_{i-1,u}$ and, thus, cover base object r_i with value $x_{i,i,u}$.
- Each base object in $\{r_i, \ldots, r_k\}$ stores the initial value.

▶ **Lemma 2.** *Let $\mathcal{A}$ be any n-process m-valued interval-solo-fast anonymous consensus algorithm. If at most k distinct base objects are written in any solo execution of $\mathcal{A}$, where $k < \min(\sqrt{n}, \Gamma^{-1}(m))$, then $\mathcal{A}$ has a k-cloning configuration.*

Proof. By induction on k, we construct a k-cloning configuration starting from the initial configuration C_0 of $\mathcal{A}$.

We divide the processes in two groups of size at most $(k^2 - k + 2)/2$ where every process in one group proposes value v and every process in the other group proposes value w. This is possible, since $k < \sqrt{n}$.

Base case. Let γ be the concatenation of executions of $\mathcal{A}$ in which, starting at C_0, a process runs in isolation until it is about to write to base object r_1 for the first time. Recall that r_1 is the first base object written in both α_v and α_w, so no process can distinguish γ from its solo execution and, thus, *gamma* is indeed an execution of $\mathcal{A}$.

It is easy to see that, since no process writes in γ, $C_1 = C_0\gamma$ is a 1-cloning configuration. Indeed, half of the processes cannot distinguish $C_0\gamma$ from $\alpha_{0,v}$ and the other half from $\alpha_{0,w}$, and all base objects are in their initial states.

As an *induction hypothesis*, consider an i-cloning configuration C_i, for some $1 \leq i < k$.

For each $u \in \{v, w\}$ we then perform the following procedure.

First for each $j = 1, \ldots, i-1$, we let one of the processes covering r_j with $x_{j,i-1,u}$ complete its write. By the induction hypothesis, there are at least $s_{j,i-1,u} \geq 1$ such processes.

Then we wake up $(k^2 - k + 2)/2 - \sum_{j=1}^{i-1} \sum_{\ell=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u}$ processes that cannot distinguish the execution from $\alpha_{i-1,u}$ and run them *lock-step* (without noticing each other) until they are about to perform their write on r_{i+1}. No such process can distinguish the execution from $\alpha_{i,u}$, and thus, we indeed obtain an execution of $\mathcal{A}$. If $\alpha_{i,u}$ contains a write on some r_m, $m = 1, \ldots, i$, after the last event of $\alpha_{i-1,u}$, then $s_{m,i,u}$ of these processes are stopped just before they perform the *last* write on r_m in $\alpha_{i+1,u}$. This can be done because $I_{m,i,u} = 1$ for every such m and $\sum_{\ell=1}^{i} \sum_{j=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u} \leq \sum_{\ell=1}^{k-1} \sum_{j=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u} = (k^2 - k)/2 < n/2$.

Let γ be the resulting extension of C_i and $C_{i+1} = C_i\gamma$ be the resulting configuration. Notice that all base objects in $\{r_{i+1}, \ldots, r_k\}$ still store the initial value in C_{i+1}.

Now consider any $j = 1, \ldots, i$ and $u = v, w$. If r_j is not written in $\alpha_{i+1,u}$, then, by the induction hypothesis and the construction of γ, r_j is covered by $s_{j,i,u} = s_{j,i-1,u} - 1$ processes writing $x_{j,i,u} = x_{j,i-1,u}$. Otherwise, by construction, r_j is covered by $s_{j,i,u}$ processes writing $x_{j,i,u}$.

Finally, for each $u \in \{v, w\}$, since $\sum_{j=1}^{i} I_{j,i,u} s_{j,i,u}$ additional processes are used to cover base objects $r_1, \ldots, r_i$, at least $(k^2 - k + 2)/2 - \sum_{\ell=1}^{i} \sum_{j=1}^{i} I_{j,\ell,u} s_{j,\ell,u}$ remaining processes cannot distinguish $C_i\gamma$ from $C_0\alpha_i$ and, thus, these processes must cover r_{i+1}.

Hence, C_{i+1} is $(i+1)$-cloning, and, by induction, $\mathcal{A}$ has a k-cloning configuration. ◀

▶ **Theorem 3.** *Any n-process m-valued interval-solo-fast anonymous consensus algorithm must have space complexity $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ and solo-write complexity $\Omega(\min(\sqrt{n}, \log m/\log\log m))$. Moreover, if the algorithm is input-oblivious, then the bounds become $\Omega(\sqrt{n})$.*

Proof. Suppose, by contradiction, that an n-process m-valued interval-solo-fast anonymous consensus algorithm uses k base objects such that $n = k^2 - k + 2$ and $k < \Gamma^{-1}(m)$.

By Lemma 2, there exists a k-cloning configuration C_k for some input values v and w. Note that in C_k, for each $u \in \{v, w\}$, every base object r_j, $j = 1, \ldots, k$, is covered by exactly $s_{j,k-1,u} = 1$ process writing value $x_{j,k-1,u}$. Also, exactly $n/2 - \sum_{\ell=1}^{k-1} \sum_{j=1}^{k-1} I_{j,\ell,u} s_{j,\ell,u} = \sum_{j=1}^{k-1}(k-j) = k(k-1)/2 + 1 - k(k-1)/2 = 1$ process cannot distinguish the execution from $\alpha_{k-1,u}$ and, thus, this process must cover r_k with $x_{k,k,u}$.

Now we take $u \in \{v, w\}$, and let the single process covering r_j, $j = 1, \ldots, k-1$ with value $x_{j,k-1,u}$ perform its write. Then we let the single process proposing u and covering r_k run solo. Notice that the process cannot distinguish the execution from $\alpha_{k,u}$ and, thus, it should eventually terminate by outputting value u.

In the resulting execution two different input values v and w are decided, implying a contradiction.

Thus, since $\Gamma^{-1}(m) = \Theta(\log m / \log \log m)$, the algorithm has a solo execution in which $\Omega(\min(\sqrt{n}, \log m / \log \log m))$ distinct base objects are written. Moreover, if the algorithm is input-oblivious, then a k-cloning configuration exists for any two values u and w, and the lower bounds become $\sqrt{n}$. ◀

▶ Remark 4. Lemma 2 shows that having at least $k^2 - k + 2$ processes is sufficient to construct a k-cloning configuration and, thus, establish a contradiction. The lower bound can be refined to $(k^2 - k)/2 + 2$ if we alternate the executions of processes proposing v with the executions of processes proposing w in each iteration of the inductive construction of C_k. Indeed, if processes proposing w were the last to execute in the construction of C_i, then every base object r_j, $j = 1, \ldots, i-1$ stores $x_{j,i-1,w}$, so in the next iteration, we may run processes proposing w first without the need to use the processes covering r_j with $x_{j,i-1,w}$. This allows us to spare half of the covering processes, implying $(k^2 - k)/2 + 2$ processes in total, which makes k closer to the upper bound $\sqrt{2n}$ we present in the next section. For the sake of simplicity, we chose to show the rougher (but asymptotically equivalent) lower bound.

4 Optimal interval-solo-fast consensus

In this section we present an algorithm that implements an interval-solo-fast consensus. This algorithm is similar to the *splitter-based* consensus algorithm in [17], except that we replace the `splitter` object with the `value-splitter` object that we introduce in this paper.

Value-splitters

A splitter provides processes with a single operation *split*() that returns a boolean response, so that (i) if a process runs solo, it must obtain *true* and (ii) *true* is returned to at most one process. A value-splitter exports a single operation *split*(v) ($v \in V$, for some input domain V) and relaxes property (ii) of splitters by allowing *multiple* processes to obtain *true* as long as they have the same input value. More precisely:

▶ **Definition 5.** A value-splitter supports a single operation, *split*() taking a parameter in V and returning a boolean response, and ensures that, for all $v, v' \in V$, and in every execution:

1. **VS-Agreement.** If invocations *split*(v) and *split*(v') return *true*, then $v = v'$.
2. **VS-Solo execution.** If a *split*(v) operation completes before any other *split*(v') operation is invoked, then it returns *true*.

We use a value-splitter object to construct an anonymous consensus algorithm. The algorithm incurs only a constant overhead with respect to the implementation of the value-splitter it uses and is interval-solo-fast assuming that the underlying value-splitter is interval-solo-fast.

Then we describe two anonymous interval-solo-fast implementations of a value-splitter. The first one is input-oblivious and exhibits $O(\sqrt{n})$ solo-write and space complexity, regardless of the number m of possible inputs. The second one exhibits complexities $O(\log m/\log\log m)$, regardless of the number of processes n. The two algorithms provide a matching upper bound to our $\Omega(\min(\sqrt{n}, \log m/\log\log m))$ lower bound.

4.1 Consensus using value-splitter

The pseudocode of our consensus algorithm is given in Algorithm 1. The value decided by the consensus is written in a variable D, initially $\perp \notin V$. The first steps by a process p are to check if D stores a non-$\perp$ value and if yes, return this value. Otherwise, the process accesses the value-splitter object VS.

If it obtains *true* from its invocation of $VS.split(v)$, p writes its input value v in a register F. Then, it reads a register Z to check if some other process has detected contention and if the value of Z is *false* (no contention) p decides its own value. Before returning the decided value, process p writes it in D. The write primitives on F and D, with a read of Z in between are intended to ensure that either process p detects that some other process is around and resorts to applying a CAS primitive on D, or the contending process adopts the input value of p.

If p obtains *false* from the value-splitter, it sets Z to *true* (contention is detected). Recall that this may happen if more than one process accessed the value-splitter, regardless of their input values. Then, p reads register F and, if F stores a non-$\perp$ value, adopts the value as its current proposal. Finally, it applies the CAS primitive on D with its proposal and decides the value read in D.

Notice that, assuming that the value-splitter is interval-solo-fast, a process running in the absence of interval contention reaches a decision applying only reads and writes.

In the following we prove that Algorithm 1 indeed implements interval-solo-fast consensus, assuming that VS is an interval-solo-fast implementation of a value-splitter. We show that such implementations exist in the next subsection.

Proofs of Algorithm 1

▶ **Lemma 6** (Agreement). *No two processes return different values.*

Proof. Given that only values written to D can be returned, it is sufficient to show that at most one value can be written in D.

By the algorithm D is updated in lines 14 and 6. Note that, since a CAS succeeds in updating the value of D in line 14 only if D stores $\perp$ and, since D is updated with a non-$\perp$ value in V, at most one process may succeed. D is updated at line 6 only if the corresponding process obtains *true* from the value-splitter. By the VS-Agreement property of value-splitters, at most one distinct value can be written in D in line 6.

Thus, the only possibility for two different values to be written in D is when one process , say p, applies a CAS in line 14 and updates D with a value v and another process writes $v' \neq v$ in D in line 6.

Note that p must have obtained *false* from the value-splitter, otherwise it would try to update D with value v. Thus, before applying CAS on D, p has read F in line 11. We

```
    Shared variables:
    $D$, $F$, initially $\bot$
    $Z$, initially $false$
    value-splitter $VS$
    Procedure: $propose(v)$
 1  if $(t := Read(D)) \neq \bot$ then return $t$
 2  ;
 3  if $VS.split(v)$ then
 4      $Write(F, v)$;
 5      if $\neg(Read(Z))$ then
 6          $Write(D, v)$;
 7          return $v$
 8      end
 9  else
10      $Write(Z, true)$;
11      if $(t := Read(F)) \neq \bot$ then $v := t$;
12      ;
13  end
14  $CAS(D, \bot, v)$;
15  $res := Read(D)$;
16  return $res$
```

Algorithm 1: Interval-solo-fast consensus

establish the contradiction by showing that p must have necessarily read v' in F and adopt it as its preferred value (line 11).

By the VS-Agreement property of value-splitters, at most one non-$\bot$ value can be found in F. Thus, since q has written v' to F in line 4, the only possible case is that p reads F before any other process writes to it. But then p has previously set the "contention flag" Z to *true* in line 10. Therefore, after q writes v' in F it must find Z set to *true* ("contention is detected") and resort to CAS instead of writing in D in line 6—a contradiction. ◀

▶ **Lemma 7** (Interval-solo-fast). *Any operation that runs in the absence of interval contention applies only reads and writes.*

Proof. If a process p invokes its *propose* operation and finds a non-$\bot$ value in D, then p returns after having applied a single read on D, so the claim follows.

Otherwise, suppose that p initially finds $D = \bot$ and applies the CAS primitive (line 14). We show that there is an operation that overlaps with the *propose* of p.

By inspecting the pseudo-code, it is easy to see that p applies the CAS primitive only if (1) it has read $Z = true$ (line 5) or (2) it has obtained *false* from VS. In both cases, by the VS-Solo Execution property, there must be another process q that has invoked $VS.split(v)$ before p has completed its *Propose* operation.

By the algorithm, before completing its operation, q writes its decided (non-$\bot$) value in D. Given that p has initially found $\bot$ in D, we deduce that the operation of q has not completed before the operation of p has started.

Thus, the two operations overlap. The assumption that the value-splitter is interval-solo-fast and the fact the algorithm contains no loops or waiting statements, implies the claim. ◀

Finally, we use Lemmata 6 and 7 to prove:

▶ **Theorem 8.** *If VS is an interval-solo-fast implementation of a value-splitter, then Algorithm 1 implements interval-solo-fast consensus with space complexity $O(k)$ and solo-write complexity $O(s)$, where k is the space complexity and s is the solo-write complexity of VS.*

The complexity claims follow directly from the pseudo-code.

4.2 Interval-solo-fast value-splitter implementations

Input-oblivious value-splitter

Algoritm 2 describes our anonymous and input-oblivious implementation of a value-splitter. The algorithm only uses an array R of k registers where $k^2 - 3k + 6 > 2n$ and is, trivially, interval-solo-fast. Thus, by Theorem 3, the space complexity of the algorithm is asymptotically optimal.

In the algorithm, a process p performing operation *split*(v) tries to write its input value to registers $R[0], \ldots, R[k-1]$. Each time, before writing to $R[i]$, p reads $i+1$ registers to verify that $R[0], \ldots, R[i-1]$ store v and $R[i]$ stores the initial value $\perp$. If this is not the case, contention is detected and the operation returns *false*. After the last write to $R[k-1]$, the operation returns *true*. Note that several processes proposing the same value and executing lock-step may return *true*.

```
Shared variables:
Array of registers $R[0 \ldots k-1]$ with $k^2 - 3k + 6 > 2n$. Initially $\perp$
Procedure: split($v$)
Lastwritten := $-1$;
while (Lastwritten $\leq k-1$) do
    $i := 0$;
    while ($i \leq$ Lastwritten) do
        if Read($R[i]$) $\neq v$ then return $false$
        ;
        $i++$;
    end
    if Read($R$[Lastwritten $+ 1$]) $\neq \perp$ then return $false$;
    ;
    Lastwritten$++$;
    Write($R$[Lastwritten], $v$);
end
return true;
B
```

Algorithm 2: Anonymous and input-oblivious value-splitter

Note also that the solo-write complexity of Algorithm 2 is $k = O(\sqrt{n})$. Since, for $i = 1$ to k, in the ith iteration, a process reads i registers, the algorithm also has optimal step complexity of $O(n)$ [1].

The following lemma will be instrumental in showing that Algorithm 2 satisfies the VS-Agreement property.

▶ **Lemma 9.** *If an execution E, two processes p and q write in $R[i]$ and $R[i+1]$ for some $0 < i < k-1$, two different values v and w, then there is a set P_i of at least i processes (different from p and q) and the following conditions are satisfied:* (1) *at the configuration that immediately succeeds the last write operation executed by processes in P_i, $R[i+1] = \perp$;* (2) *E passes through a configuration C such that $R[i] \neq \perp$ in C and each process in P_i executes exactly one write operation after C.*

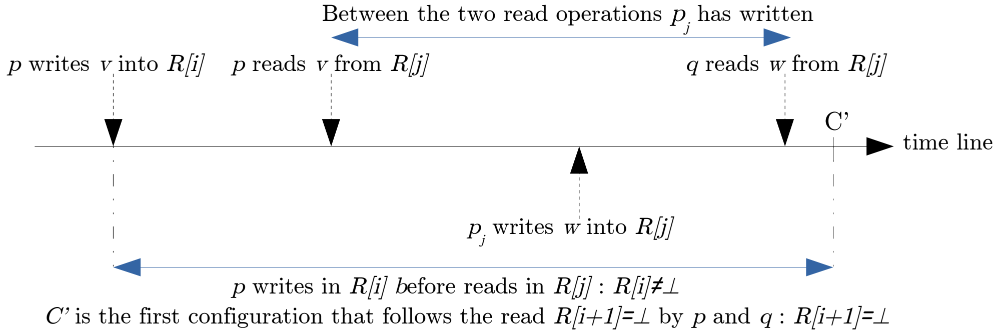

Between the two read operations p_j has written
p writes v into $R[i]$
p reads v from $R[j]$
q reads w from $R[j]$
C'
time line
p_j writes w into $R[j]$
p writes in $R[i]$ before reads in $R[j]$: $R[i]\neq\perp$
C' is the first configuration that follows the read $R[i+1]=\perp$ by p and q : $R[i+1]=\perp$

Figure 3 Execution for Lemma 9, assuming that p reads $R[j]$ before q.

Proof. Fix an i such that $0 < i < k-1$ and let p and q be two processes that write, respectively, values v and w in both $R[i]$ and $R[i+1]$, where $v \neq w$.

By the pseudocode of Algorithm 2, before writing in $R[i+1]$, a process reads $R[0], R[1], \ldots, R[i+1]$, and the value it reads from $R[j]$ is its input value for $0 \leq j \leq i$ and the initial value for $j = i+1$.

Consider the sequences of read operations executed by p and q, respectively, after their write in $R[i]$ and before writing in $R[i+1]$. Let C' be the configuration immediately after both p and q perform their reads of $R[i+1]$ that return $\perp$ in E. By the algorithm, writes in $R[i+1]$ by both p and q follow C' in E.

Also, since for each $j = 0, \ldots, i-1$ p reads v in $R[j]$ and q reads w in $R[j]$, there is a process p_j that has written in $R[j]$ between these two read operations. We show that this is the last write of p_j. Indeed, before performing the next write (on $R[j+1]$), p_j reads all registers and in particular it will read $R[i]$, where $i > j$. Since the write by p_j follows the read on $R[j]$ either by process p or by process q, it follows the write into $R[i]$ by the corresponding process. Thus, in the configuration immediately before the write into $R[j]$ by p_j we have $R[i] \neq \perp$. The check in line 9 implies that p_j cannot write to any register after $R[j]$. Note that p_j must be different from p and q: otherwise, we contradict the fact that both p and q write in $R[i]$, $i > j$.

Finally, since the last write operation of p_j preceeds configuration C', at the configuration immediately after this write $R[i+1]$ stores the initial value. This is illustrated in Figure 3 for the case when p reads $R[j]$ before q. Moreover, for each $j, \ell \in \{0, 1, \ldots i-1\}$ with $j \neq \ell$, $p_j \neq p_l$. Thus, the set P_i of i processes p_j, $j = 1, \ldots, i-1$, satisfies the two conditions of the lemma. ◀

▶ **Lemma 10** (VS-Agreement). *If invocations split(v) and split(v') return true, then $v = v'$.*

Proof. Suppose, by contradiction, that *split*(v) invoked by process p and *split*(w) invoked by process q both return *true* with $v \neq w$. Recall that a process has to write its input value in all the registers to return *true*. Then for each $0 \leq i \leq k-1$, p and q have written in register $R[i]$ the value v and w respectively. For each $i = 1, \ldots k-2$, let P_i be the i processes, as defined in Lemma 9.

Consider any two set P_i, P_j, $0 < i < j < k-1$. We show that $P_i \cap P_j = \emptyset$. Indeed, by the definition of P_i, in the configuration when the processes in P_i have completed all their writes, $R[i+1]$ stores $\perp$ and, by the algorithm, since $j > i$, $R[j]$ also stores $\perp$. But, by the definition of P_j, each process in P_j has executed a write operation after a configuration where $R[j] \neq \perp$. Thus, P_i and P_j are disjoint.

Recall p and q write to $R[k-1]$ and, thus, do not belong to $\cup_{i=1}^{k-2} P_i$. Hence, we have at least $2 + \sum_{i=1}^{k-2} i = 2 + \frac{k^2-3k+2}{2}$ processes in total, which contradicts the hypothesis that $k^2 - 3k + 6 > 2n$. ◀

▶ **Theorem 11.** *Algorithm 2 is an interval-solo-fast anonymous input-oblivious implementation of a value-splitter with solo-write and space complexities in $O(\sqrt{n})$.*

Proof. Since only read-write registers are used, the algorithm is trivially interval-solo-fast.

By Lemma 10, the algorithm satisfies the *VS-Agreement* property. We prove in the following that the *VS-Solo execution* property is also satisfied. Consider any solo execution E in which a *split*(v) by a process p completes and suppose, by contradiction, that the operation returns $false$. By inspecting the pseudocode, it is easy to see that the value of *Lastwritten* is equal to the index of the last register p wrote or to -1 if no such writes exists. To return $false$ p has either read a value different from its input (line 5) or a value different from $\perp$ in a register p has not yet written (line 9). But this contradicts the fact that E is a solo execution. Thus, the algorithm satisfies the Solo-Execution property of value-splitters. ◀

Non-input-oblivious value-splitter

For completeness, we briefly describe an anonymous value-splitter algorithm based on earlier work [1] that exhibits $O(\log m / \log\log m)$ complexity.

A trivial adaptation of the `weak conflict-detector` proposed in [1] implements an interval-solo-fast value-splitter. A weak conflict-detector exports a single operation *check*(v) with an input v and return *true* (conflict is detected) or *false* (no conflict is detected). If no two operations are invoked with different inputs, then no operation returns *true*, otherwise, at least one operation returns *false*.

Our value-splitter implementation presented in Algorithm 3 is obtained by the weak conflict-detector algorithm in [1], where the output is determined as the negation of the outcome of the weak conflict-detector.

```
Shared variables:
Registers $R[1..k]$, initially $\perp$
Procedure: $split(v)$
for $i := 1..k$ do
    $t := Read(R[\pi_v(i)])$;
    if $t = \perp$ then $Write(R[\pi_v(i)], v)$;
    ;
    if $t \neq v$ then return $false$;
    ;
end
return $true$;
```

Algorithm 3: Non-input-oblivious value-splitter

The algorithm uses an array R of k registers, where $k! = m$. Each input value v of a *split* operation determines a unique permutation π_v of the registers in R that is used as the order in which the processes access the registers. Therefore, the algorithm is not input-oblivious. In its i-th access, a process executing *split*(v) first reads register $R[\pi_v(i)]$; if $\perp$ is read, the process writes v to it; If a value $v' \neq v$ is read, it returns *false* (contention is detected). If the process succeeds in writing v in all registers prescribed by π_v, it returns *true*. The algorithm is also trivially anonymous and interval-solo-fast.

▶ **Theorem 12.** *Algorithm 3 implements anonymous interval-solo-fast m-valued value-splitter with solo-write and space complexity in $O(\log m/\log\log m)$.*

Proof. If an operation *split*(v) runs solo, then no value other than v can be found in any $R[\pi_v(i)]$ (line 2). Thus the *VS-Solo Execution* property is ensured.

Suppose, by contradiction, that two operations, *split*(v), performed by p_v, and *split*(v'), performed by $p_{v'}$, return *true*. Let j, ℓ be two indexes in $\{1, \dots, k\}$ such that j appears before ℓ in π_v but ℓ appears before j in $\pi_{v'}$. By the algorithm, before returning *true*, p_v and $p_{v'}$ have read, respectively, v and v' in both $R[j]$ and $R[\ell]$.

Without loss of generality, let v be written to $R[j]$ before v' is written to $R[\ell]$. By the algorithm, before any process performing *split*(v') reads $R[j]$ in line 2 (and, thus, writes v' to $R[j]$ in line 4), v' has been written to $R[\ell]$, and, by the assumption, v has been written to $R[j]$. Hence, the process will not find $\perp$ in $R[j]$ and will not write to $R[\ell]$—a contradiction. Therefore, the algorithm satisfies the VS-Agreement property.

Since every operation performs k writes and k reads, where $k! = m$, the step and space complexities of the algorithm are $O(\log m/\log\log m)$. ◀

5 Concluding remarks

In this paper, we present matching lower and upper bounds $\Theta(\min(\sqrt{n}, \log m/\log\log m))$ on the space and solo-write complexity of anonymous interval-solo-fast consensus, which appears to be one of the first non-trivial tight bound for consensus, along with a concurrent result on the space complexity of solo-terminating anonymous consensus [8]. Given that *non-anonymous* interval-solo-fast algorithms can be achieved with only constant space and step complexities [17], our results exhibits a complexity gap between anonymous and non-anonymous consensus. The proof of our lower bound is based on constructing executions in which no process is aware of interval contention and, thus, the lower bounds also apply to *abortable* [2, 11] consensus algorithms, where operations are allowed to return a specific *abort* response when interval contention is detected, and be-reinvoked later. An interesting open question is whether a matching abortable consensus algorithm can be found.

References

1 James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014. doi:10.1007/s00224-013-9448-1.

2 Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.

3 Zohir Bouzid, Pierre Sutra, and Corentin Travers. Anonymous agreement: The janus algorithm. In *Principles of Distributed Systems – 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 175–190, 2011. doi:10.1007/978-3-642-25873-2_13.

4 Harry Buhrman, Juan A. Garay, Jaap-Henki Hoepman, and Mark Moir. Long-lived renaming made fast. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 194–203, 1995.

5 Claire Capdevielle, Colette Johnen, Petr Kuznetsov, and Alessia Milani. Brief Announcement: On the Uncontended Complexity of Anonymous Consensus. In *DISC*, October 2015.

6 Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, September 1998.

7 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.

8 Rati Gelashvili. On the Optimal Space Complexity of Consensus for Anonymous Processes. In *DISC*, October 2015.

9 George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An o(sqrt n) space bound for obstruction-free leader election. In *Distributed Computing – 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 46–60, 2013. doi:10.1007/978-3-642-41527-2_4.

10 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007. doi:10.1007/s00446-007-0042-0.

11 Vassos Hadzilacos and Sam Toueg. On deterministic abortable objects. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC'13, pages 4–12, New York, NY, USA, 2013. ACM. doi:10.1145/2484239.2484241.

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

13 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

14 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

15 Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, January 1987.

16 M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

17 Victor Luchangco, Mark Moir, and Nir Shavit. On the uncontended complexity of consensus. In FaithEllen Fich, editor, *Distributed Computing*, volume 2848 of *Lecture Notes in Computer Science*, pages 45–59. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-39989-6_4.

18 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, October 1995. doi:10.1016/0167-6423(95)00009-H.

The Relative Power of Composite Loop Agreement Tasks*

Vikram Saraph[1] and Maurice Herlihy[2]

1 **Department of Computer Science, Brown University, Providence, USA**
vsaraph@cs.brown.edu

2 **Department of Computer Science, Brown University, Providence, USA**
mph@cs.brown.edu

Abstract

Loop agreement is a family of distributed tasks that includes set agreement and simplex agreement, and was used to prove the undecidability of wait-free solvability of distributed tasks by read/write memory. Herlihy and Rajsbaum defined the algebraic signature of a loop agreement task, which consists of a group and a distinguished element. They used the algebraic signature to characterize the relative power of loop agreement tasks. In particular, they showed that one task implements another exactly when there is a homomorphism between their respective signatures sending one loop to the other. In this paper, we extend the previous result by defining the *composition* of multiple loop agreement tasks to create a new one with the same combined power. We generalize the original algebraic characterization for relative power to compositions of tasks. In this way, we can think of loop agreement tasks in terms of their basic building blocks. We also investigate a category-theoretic perspective of loop agreement by defining a category of loops, showing that the algebraic signature is a functor, and proving that our definition of task composition is the "correct" one, in a categorical sense.

1998 ACM Subject Classification F.1.2 [Modes of Computation] Parallelism and Concurrency

Keywords and phrases Distributed computing, loop agreement, task composition, topology

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.13

1 Introduction

Characterizing the relative power of synchronization primitives is one of the fundamental questions in distributed computing. It lies at the heart of processor instruction set architectures (is it better to provide compare-and-swap or test-and-set?), middleware frameworks (message-passing or atomic broadcast?), and similar design problems. Given two synchronization primitives, we can ask whether one is more powerful than the other, or whether their *composition* is more powerful then either individually. Such questions often arise for *subconsensus* tasks [12], synchronization primitives that have no wait-free implementations using read-write memory, but that are not strong enough to solve consensus. Although some partial results are known, for example, that renaming is strictly weaker than set agreement [5], no general technique is known for evaluating the computational power of compositions of subconsensus primitives.

In this paper, we report some progress in understanding the power of composition for an important family of subconsensus tasks: the *loop agreement* tasks [8]. We define a natural

* Supported by NSF grant 1301924.

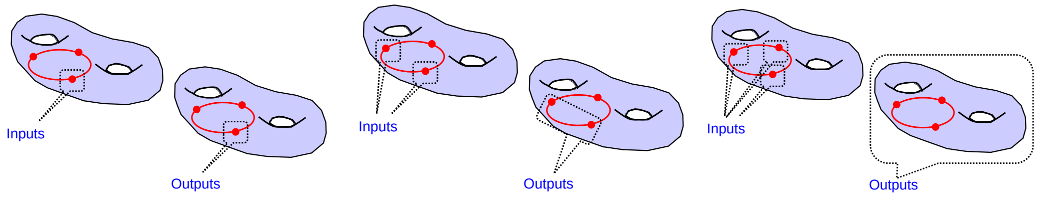


Figure 1 Loop Agreement.

notion of composition for these tasks, characterize the composition's computational power in terms of algebraic structures, and show that there is a non-obvious but natural sense in which loop agreement tasks can be considered to be closed under composition.

A *task* is a distributed problem in which each process begins with an input, communicates with others, and returns an output according to the task's specification. Common examples of tasks include consensus [3], set agreement [2], and renaming [1]. *Protocols* are distributed programs that solve tasks. A protocol is *wait-free* if every non-faulty process running the protocol eventually finishes execution, regardless of other process failures. One task *implements* another if a protocol for the first task can be modified in a simple way to solve the second task. In this paper, we will assume we are given "black box" implementations of certain tasks which we will use to construct protocols for other tasks.

Loop agreement [8] is a family of tasks that requires processes to rendezvous along a loop in a given space. As illustrated in Fig. 1, we are given a topological space, a loop in that space, and three distinguished points on the loop. Very informally, each participating process starts on one of these distinguished points. If all processes start on the same point, they all halt on that point (Fig. 1a). If they all start on two distinct distinguished points,, then they converge to "nearby" points along the path linking their starting points (Fig. 1b). Finally, if they start on all three distinguished points, then they converge to "nearby" points anywhere in the space (Fig. 1c).

Each loop agreement task has an *algebraic signature* [9] given by a group G and an element $g \in G$. If tasks T_1 and T_2 have signatures (G_1, g_1) and (G_2, g_2), respectively, then T_1 implements T_2 exactly when there is a group homomorphism $\phi : G_1 \to G_2$ mapping g_1 to g_2, so the operational problem of loop agreement tasks implementing one another is reduced to an algebraic characterization.

In this paper, we define a notion of composition for loop agreement tasks and characterize how such compositions can implement other loop agreement tasks. Roughly speaking, the *composition* of n loop agreement tasks is a task in which each process solves each of the n tasks in parallel. We show that tasks $\{T_i\}$ with signatures $\{(G_i, g_i)\}$ solve T with signature (G, g) if and only if there is a homomorphism $\phi : G_1 \times \cdots \times G_n \to G$ mapping $(g_1, \ldots, g_n)$ to g. We also provide a means of replacing the loop agreement tasks $\{T_i\}$ with an equivalent task $\prod T_i$, called the *composition* of the $\{T_i\}$. This composition of tasks is also a loop agreement task, and has relative power equivalent to that of all the $\{T_i\}$. That is, we can construct a protocol for $\prod T_i$ given "black box" implementations of each of the T_i, and vice-versa: we can construct a protocol for any of the T_i from $\prod T_i$.

Finally, we can use elementary category theory to provide evidence that we have the "correct" notion of task composition. We define a category of loop agreement tasks, **Loop**, and show that the map assigning tasks to algebraic signatures is a functor into the category of pointed groups, **pGrp**. We also show that composition of loop agreement tasks is the categorical product in **Loop**, which strongly suggests that composition of tasks as defined in this paper is a natural way to capture the operational notion of parallel composition.

2 Related Work

Herlihy and Shavit [10, 11] introduced the use of algebraic and combinatorial topology to prove impossibility results. Gafni and Koutsoupias [4] were the first to use the fundamental group to show the undecidability of wait-free solvability of certain tasks. Herlihy and Rajsbaum [8, 9] extended the undecidability results to other models, introducing the family of loop agreement tasks and their algebraic signatures.

Loop agreement has been generalized to higher dimensions. Liu, Xu, and Pan [16] define *n-rendezvous tasks*, where processes begin on distinguished vertices of an embedded $(n-1)$-sphere of an n-dimensional complex, and converge on a simplex of the embedded sphere. They generalize the algebraic signature characterization to a subclass of rendezvous tasks called *nice* rendezvous tasks, which are tasks whose output complexes have trivial homology groups below and above dimension n, and a free Abelian n-th homology group. The authors apply their main result to show there are countably infinitely many inequivalent nice rendezvous tasks.

Liu, Pu, and Pan [15] explore a lower-dimensional variant of loop agreement called *degenerate loop agreement*, which unlike loop agreement includes binary consensus. Processes begin on a 1-dimensional complex, or a graph, and must converge to one of two possible starting locations in the graph. The authors prove that there are only two inequivalent tasks degenerate tasks: the trivial task and binary consensus.

3 Background

In the first subsection, we describe the mathematical model used for distributed tasks, of which more details can be found in Herlihy, Kozlov, and Rajsbaum [7]. In the second subsection, we summarize important definitions and results from algebraic topology.

3.1 Distributed Computing

Formally, a (colorless) *task* is a triple $(\mathcal{I}, \mathcal{O}, \Gamma)$, where objects $\mathcal{I}$ and $\mathcal{O}$, called the *input* and *output complexes* of the task, are mathematical structures known as simplicial complexes. A *simplicial complex* on a set V is a collection of subsets $\mathcal{C}$ of V such that $\mathcal{C}$ is downward closed under the subset relation. Complexes can be thought of as higher-dimensional graphs where “edges” may “connect” more than two vertices. In the context of tasks, vertices of $\mathcal{I}$ represent process input values, while simplexes of $\mathcal{I}$ represent valid input combinations. Likewise, vertices of $\mathcal{O}$ represent process output (or decision) values, and simplexes represent valid output combinations. Relating $\mathcal{I}$ and $\mathcal{O}$ is the map $\Gamma : \mathcal{I} \to 2^{\mathcal{O}}$, which is called the task’s *specification map*, and carries simplexes of $\mathcal{I}$ to subcomplexes of $\mathcal{O}$ in a monotonic way[1]. The map Γ associates each input combination with a set of legal output combinations.

Protocols are objects that solve tasks, and are also modeled by triples $(\mathcal{I}, \mathcal{P}, \Xi)$. As with tasks, $\mathcal{I}$ is the protocol’s *input complex*. The object $\mathcal{P}$ is also a simplicial complex, which is called the *protocol complex*, and is similar to a task’s output complex, but has a slightly different meaning. Rather than a final decision value, a vertex in $\mathcal{P}$ represents a process’s uninterpreted state (or view) after running the protocol. The map $\Xi : \mathcal{I} \to 2^{\mathcal{P}}$, called the *execution map*, is monotonic, and represents the possible states in which processes may result after running the protocol.

[1] In general, if $\mathcal{A}$ and $\mathcal{B}$ are simplicial complexes, then a function $\Phi : \mathcal{A} \to 2^{\mathcal{B}}$ is called a *carrier map* if for each $\sigma \subseteq \tau \in \mathcal{A}$, $\Phi(\sigma)$ is a simplicial complex, and $\Phi(\sigma) \subseteq \Phi(\tau)$ (or Φ is *monotonic*).

A *simplicial map* $\delta : \mathcal{I} \to \mathcal{O}$ between two complexes is a vertex map that sends simplexes to simplexes; that is, $\delta(\sigma) \in \mathcal{O}$ for each $\sigma \in \mathcal{I}$. A protocol $(\mathcal{I}, \mathcal{P}, \Xi)$ *solves* $(\mathcal{I}, \mathcal{O}, \Gamma)$ if there exists a simplicial map $\delta : \mathcal{O} \to \mathcal{P}$, called a *decision map*, that respects the task specification Γ. Formally, δ respects Γ if for each simplex $\sigma \in \mathcal{I}$, we have $(\delta \circ \Xi)(\sigma) \subseteq \Gamma(\sigma)$.

Some tasks are inherently harder than others, and sometimes we can transform a protocol for one task into a protocol for another. We say task T_1 *implements* T_2 if we can use the output complex of T_1 (or a subdivision of it) as a protocol complex for solving T_2. Mathematically speaking, if $T_1 = (\mathcal{I}, \mathcal{O}_1, \Gamma_1)$ and $T_2 = (\mathcal{I}, \mathcal{O}_2, \Gamma_2)$, then T_1 implements T_2 if there exists a natural number N and a simplicial map $\phi : \mathrm{Bary}^N(\mathcal{O}_1) \to \mathcal{O}_2$ such that $(\phi \circ \mathrm{Bary}^N \circ \Gamma_1)(\sigma) \subseteq \Gamma_2(\sigma)$ for each $\sigma \in \mathcal{I}$. The barycentric subdivision operator Bary is a topological operator (see the next section) that models read/write memory. Two tasks are *equivalent* if they implement each other.

3.2 Algebraic Topology

Before we can define loop agreement, we must briefly introduce the relevant machinery from algebraic topology. We assume a basic understanding of point-set topology. The algebraic topology used is at the undergraduate level, of which a formal treatment can be found in Hatcher [6]. We begin with the formal definition of a simplicial complex.

3.2.1 Simplicial Complexes

▶ **Definition 1.** Let V be any set, whose elements are called *vertices*. A *simplicial complex* (over V) is a set of subsets $\mathcal{C}$ of V such that for each set $\tau \in \mathcal{C}$, if $\sigma \subseteq \tau$, then $\sigma \in \mathcal{C}$. That is, $\mathcal{C}$ is downward closed under taking subsets. Elements of $\mathcal{C}$ are called *simplexes*.

We can think of simplicial complexes as a generalization of graphs, where simplexes may be incident to more than two vertices. Graphs are then precisely the simplicial complexes whose simplexes contain at most two vertices. Nontrivial graphs have dimension 1, and in general, the *dimension* of a complex $\mathcal{C}$ is $n - 1$, where n is the size of the largest simplex in $\mathcal{C}$. The dimension of a simplex σ is simply $|\sigma| - 1$. The *standard n-simplex*, Δ^n, is the simplicial complex on $n + 1$ vertices containing all possible simplexes. By convention, we will use $\{0, \dots, n\}$ for the vertex set of Δ^n.

A *subcomplex* of $\mathcal{C}$ is a subset $\mathcal{B} \subseteq \mathcal{C}$ that is also a simplicial complex. For each nonnegative integer k, the *k-skeleton* of $\mathcal{C}$, denoted $\mathrm{skel}^k(\mathcal{C})$, is the subcomplex of $\mathcal{C}$ containing all simplexes of dimension at most k.

So far, simplicial complexes are purely combinatorial, but they can also be realized as topological spaces. Notationally, if $\mathcal{C}$ is a complex, then its geometric realization is denoted by $|\mathcal{C}|$. As previously mentioned, the *barycentric subdivision* is an operator that models read/write memory, and is better understood geometrically than combinatorially. Given a geometric simplicial complex $|\mathcal{C}|$, we can create another geometric simplicial complex by adding new vertices to the barycenter of each simplex, and adding new simplexes accordingly. This gives rise to an abstract simplicial complex, denoted $\mathrm{Bary}(\mathcal{C})$. Notice that the barycentric subdivision does not change the geometry of the original complex; that is, $|\mathrm{Bary}(\mathcal{C})| = |\mathcal{C}|$.

The barycentric subdivision is an important tool in approximating continuous functions with simplicial maps. If $f : |\mathcal{A}| \to |\mathcal{B}|$ is a continuous function between complexes, then a simplicial map $\phi : \mathcal{A} \to \mathcal{B}$ is called a *simplicial approximation* of f if for every $p \in |\mathcal{A}|$, $|\phi|(p)$ is contained in the smallest simplex containing $f(p)$. Using the barycentric subdivision, we can construct a simplicial approximation of any continuous function, as stated below.

▶ **Fact 2** (Simplicial Approximation). *Let $f : |\mathcal{A}| \to |\mathcal{B}|$ be a continuous function between simplicial complexes. Then there exists an $N \in \mathbb{N}$ and a simplicial map $\phi : Bary^N(\mathcal{A}) \to \mathcal{B}$ that is a simplicial approximation of f.*

We can take products of simplicial complexes. The product of two complexes is another complex that combines the structures of the original two.

▶ **Definition 3.** Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be simplicial complexes, and let $V(\mathcal{C}_1)$ and $V(\mathcal{C}_2)$ be their vertex sets, respectively. Then the *(categorical) product of simplicial complexes* is a complex $\mathcal{C}_1 \times \mathcal{C}_2$ with vertex set $V(\mathcal{C}_1) \times V(\mathcal{C}_2)$. A subset σ of $V(\mathcal{C}_1) \times V(\mathcal{C}_2)$ is a simplex in $\mathcal{C}_1 \times \mathcal{C}_2$ if and only if $\rho_1(\sigma)$ and $\rho_2(\sigma)$ are simplexes in $\mathcal{C}_1$ and $\mathcal{C}_2$, where ρ_1 and ρ_2 are projections onto the first and second coordinates, respectively.

Intuitively, the product of complexes is a way of combining two complexes in the "best possible way," and operationally, the product captures all possible combinations of process views if two tasks are solved in parallel. It is an important technical point that the product of complexes and product of topological spaces are not the same; it is not true that $|\mathcal{A}| \times |\mathcal{B}|$ and $|\mathcal{A} \times \mathcal{B}|$ are homeomorphic[2]. They are, however, "homotopy equivalent," which is a type of equivalence described in the next section.

To each topological space we can assign an invariant called the fundamental group, a basic construct taken from algebraic topology. The fundamental group is used to define the algebraic signature of a loop agreement task.

3.2.2 Homotopy and the Fundamental Group

Given a topological space X and a base point $x_0 \in X$, a *loop* in X based at x_0 is a continuous function $\lambda : [0,1] \to X$ such that $\lambda(0) = \lambda(1) = x_0$. Two loops λ_1 and λ_2 based at x_0 are (loop) *homotopic* if one loop can be continuously deformed to the other. More precisely, λ_1 and λ_2 are homotopic if there is a continuous function $H : [0,1] \times [0,1] \to X$ such that $H(0,-) = \lambda_1$, $H(1,-) = \lambda_2$, and $H(-,0) = H(-,1) = x_0$. Homotopy is an equivalence relation. We write $[\lambda]$ to denote the equivalence class of all loops homotopic to λ.

Let $\alpha : [0,1] \to X$ and $\beta : [0,1] \to X$ be two loops based at x_0. Then we can *concatenate* α and β to get another loop, $\alpha \cdot \beta$, defined by traversing α, returning to x_0, and then traversing β. The loop $\alpha \cdot \beta : [0,1] \to X$, also based at x_0, is defined as

$$(\alpha \cdot \beta)(t) = \begin{cases} \alpha(2t) & \text{for } 0 \le t \le \frac{1}{2} \\ \beta(2t-1) & \text{for } \frac{1}{2} \le t \le 1 \end{cases}$$

Concatenation behaves well with homotopy. If α and β are homotopic to α' and β', respectively, then $[\alpha \cdot \beta] = [\alpha' \cdot \beta']$. From this it follows that concatenation is associative on classes of loops based at x_0. In fact, concatenation is a group operation on classes of loops based at x_0, with the inverse computed by traversing a loop in the opposite direction, and the identity element being the class of all loops homotopic to the constant loop at x_0. Formally, the inverse of $[\alpha]$ is the class of the loop $\alpha^{-1}(t) = \alpha(1-t)$, and the class $[e]$ of loop $e(t) = x_0$ serves as the identity.

▶ **Definition 4.** Let X be a topological space, and let $x_0 \in X$ be a base point. Then the *fundamental group* of X at x_0, denoted $\pi_1(X, x_0)$, is the set of all loop homotopy classes with

[2] For example, $|\Delta^1| \times |\Delta^1|$ is homeomorphic to a square, but $|\Delta^1 \times \Delta^1|$ is homeomorphic to a tetrahedron.

concatenation as its group operation. If X is path-connected, then $\pi_1(X, x_0)$ is independent of x_0, and we simply write $\pi_1(X)$.

If $f : (X, x_0) \to (Y, y_0)$ is a base point-preserving continuous function, then π_1 also induces a group homomorphism $f_* : \pi_1(X, x_0) \to \pi_1(Y, y_0)$ called the *induced homomorphism*, defined by $f_*([\lambda]) = [f \circ \lambda]$.

Henceforth, we assume all topological spaces and simplicial complexes under consideration are path-connected. For brevity, if $\mathcal{C}$ is a complex, we write $\pi_1(\mathcal{C})$ instead of $\pi_1(|\mathcal{C}|)$. An important property of the fundamental group is how it behaves with the product of topological spaces.

► **Fact 5.** *Let X and Y be topological spaces. Then $\pi_1(X \times Y) \cong \pi_1(X) \times \pi_1(Y)$.*

Homotopy is defined for loops, but it is more generally defined for continuous functions. Two continuous functions $f, g : X \to Y$ are *homotopic* if there is a continuous $H : X \times [0, 1] \to Y$ such that $H(-, 0) = f$ and $H(-, 1) = g$. We write $f \simeq g$ if this is the case. If in addition $X \subseteq Y$ and H fixes X, then H is called a *deformation retraction* and we say Y *deformation retracts* onto X. If δ is a simplicial approximation of a continuous function h, then it is known that $|\delta| \simeq h$.

Using homotopy, we can define a weak equivalence between topological spaces called homotopy equivalence.

► **Definition 6.** Let X and Y be topological spaces. Then X and Y are *homotopy equivalent*, or $X \simeq Y$, if there are continuous functions $f : X \to Y$ and $g : Y \to X$ such that $g \circ f \simeq \mathrm{id}_X$ and $f \circ g \simeq \mathrm{id}_Y$. The maps f and g are called *homotopy equivalences* and are *homotopy inverses* of one another.

Homeomorphic spaces are clearly homotopy equivalent. Homotopy equivalent spaces have the same fundamental group.

► **Fact 7.** *Let X and Y be topological spaces. If $X \simeq Y$, then $\pi_1(X) \cong \pi_1(Y)$.*

The next few facts are specifically about simplicial complexes. Recall that given two simplicial complexes $\mathcal{A}$ and $\mathcal{B}$, $|\mathcal{A}| \times |\mathcal{B}|$ and $|\mathcal{A} \times \mathcal{B}|$ are not topologically equivalent, though they are homotopy equivalent. See Kozlov's book on combinatorial algebraic topology for a detailed proof of this result [13].

► **Fact 8.** *Let $\mathcal{A}$ and $\mathcal{B}$ be simplicial complexes. Then $|\mathcal{A}| \times |\mathcal{B}| \simeq |\mathcal{A} \times \mathcal{B}|$.*

It follows that $|\mathcal{A}| \times |\mathcal{B}|$ and $|\mathcal{A} \times \mathcal{B}|$ have the same fundamental group. This will allow us to pass between the categorical product of $\mathcal{A}$ and $\mathcal{B}$ and the topological product of $|\mathcal{A}|$ and $|\mathcal{B}|$.

► **Fact 9.** *Let $\mathcal{C}$ be a complex. Then the inclusion $\iota : skel^2(\mathcal{C}) \to \mathcal{C}$ induces an isomorphism on fundamental groups.*

This fact can be derived from the following, more general result, which can be found in Hatcher [6]. We call a continuous function $g : |\mathcal{A}| \to |\mathcal{B}|$ *cellular* if g maps skeletons to skeletons, or more precisely, if $g(|\mathrm{skel}^n(\mathcal{A})|) \subseteq |\mathrm{skel}^n(\mathcal{B})|$ for every n. Then every continuous $f : |\mathcal{A}| \to |\mathcal{B}|$ is homotopic to such a map g, as seen below.

► **Fact 10** (Cellular Approximation). *Let $f : |\mathcal{A}| \to |\mathcal{B}|$ be a continuous function between simplicial complexes $\mathcal{A}$ and $\mathcal{B}$. Then f is homotopic to a cellular function $g : |\mathcal{A}| \to |\mathcal{B}|$. Furthermore, if $\mathcal{C} \subseteq \mathcal{A}$ is a subcomplex such that f is already cellular on $|\mathcal{C}|$, then we may require the homotopy between f and g to fix $|\mathcal{C}|$.*

Now suppose we have a homotopy on a subcomplex and we want to extend it to the entire simplicial complex. The next fact, also found in Hatcher [6], allows us to do this.

▶ **Fact 11** (Homotopy Extension). *Let $\mathcal{C} \subseteq \mathcal{A}$ and $\mathcal{B}$ be simplicial complexes, and let $F : |\mathcal{A}| \to |\mathcal{B}|$ be a continuous function. Suppose we have a homotopy $H : |\mathcal{C}| \times [0,1] \to |\mathcal{B}|$ such that $H(-,0) = F|_{|\mathcal{C}|}$. Then there is a homotopy extending H to all of $|\mathcal{A}|$, respecting F. That is, we can find homotopy $H' : |\mathcal{A}| \times [0,1] \to |\mathcal{B}|$ such that $H'|_{|\mathcal{C}|\times[0,1]} = H$ and $H'(-,0) = F$.*

3.3 Loop Agreement

We need a few more definitions before introducing loop agreement tasks.

▶ **Definition 12.** Let $\mathcal{C}$ be a simplicial complex. An *edge path* in $\mathcal{C}$ is an alternating sequence of vertices and edges, $v_1, e_1, v_2, e_2, \ldots, v_{k-1}, e_{k-1}, v_k$, where $e_i = \{v_i, v_{i+1}\}$. An *edge loop* is an edge path with $v_0 = v_k$.

▶ **Definition 13.** Let $\mathcal{C}$ be a simplicial complex. Then a *triangle loop* in $\mathcal{C}$ is a six-tuple $\lambda = (v_0, v_1, v_2, p_{01}, p_{12}, p_{20})$ such that each v_i is a vertex in $\mathcal{C}$ and p_{ij} is an edge path between v_i and v_j.

Triangle loops are indeed loops in the topological sense, but they can also be viewed as subcomplexes with designated vertices and edge paths.

▶ **Definition 14.** A *loop agreement task* is a task $(\mathcal{I}, \mathcal{O}, \Gamma)$ for which $\mathcal{I}$ is the standard 2-simplex, $\mathcal{O}$ is a (path-connected) 2-dimensional simplicial complex with triangle loop $\lambda = (v_0, v_1, v_2, p_{01}, p_{12}, p_{20})$, and Γ is defined as:

$$\Gamma(\sigma) = \begin{cases} \{v_i\} & : \sigma = \{i\} \\ p_{ij} & : \sigma = \{i, j\} \\ \mathcal{O} & : \sigma = \{0, 1, 2\} \end{cases}$$

Notationally, we write Loop$(\mathcal{O}, \lambda)$. Input vertices are carried to the designated vertices of λ, the input edges are carried to paths between designated vertices, and the input triangle is carried to the whole output complex. The *algebraic signature* of Loop$(\mathcal{O}, \lambda)$ is $(\pi_1(\mathcal{O}), \lambda)$, and is used in the main theorem by Herlihy and Rajsbaum [9]:

▶ **Theorem 15** (Herlihy and Rajsbaum). *Task Loop$(\mathcal{K}_1, \lambda_1)$ implements Loop$(\mathcal{K}_2, \lambda_2)$ if and only if there exists a group homomorphism $h : \pi_1(\mathcal{K}_1) \to \pi_1(\mathcal{K}_2)$ such that $h([\lambda_1]) = [\lambda_2]$.*

4 Composite Loop Agreement

We now present the main contribution of this paper: parallel composition of loop agreement tasks and the characterization of their relative power.

4.1 Implementation by Multiple Tasks

Informally, to implement one task by several others, we run protocols for each implementing task and use the combined output as a protocol complex. Given two loop agreement tasks, the composite task's output complex is the 2-skeleton of the product of their output complexes, and the composite task's loop is the "diagonal" of the product of the two original loops.

▶ **Definition 16.** Let $\lambda_1 = (v_0, v_1, v_2, p_{01}, p_{12}, p_{20})$ and $\lambda_2 = (w_0, w_1, w_2, q_{01}, q_{12}, q_{20})$ be triangle loops in complexes $\mathcal{A}$ and $\mathcal{B}$, respectively. Then the *diagonal product* of λ_1 and λ_2, denoted $\lambda_1 \star \lambda_2$, is the triangle loop $(u_0, u_1, u_2, r_{01}, r_{12}, r_{20})$ in $\mathcal{A} \times \mathcal{B}$, where $u_i = (v_i, w_i)$. The path r_{ij} is defined by traversing p_{ij} while w_i is fixed, followed by traversing q_{ij} while v_j is fixed. Note that we will use $p_{ij} \star q_{ij}$ to denote the path defined by r_{ij} as above, though strictly speaking, the $\star$ operator denotes two different operations in $\lambda_1 \star \lambda_2$ and $p_{ij} \star q_{ij}$.

▶ **Definition 17.** Let $T_1 = \text{Loop}(\mathcal{K}_1, \lambda_1)$, $T_2 = \text{Loop}(\mathcal{K}_2, \lambda_2)$, and $T = \text{Loop}(\mathcal{K}, \lambda)$ be loop agreement tasks. Let Γ_1, Γ_2, and Γ be their respective specification maps. We say T_1 and T_2 *implement* T if there is an $N \in \mathbb{N}$ and a simplicial map $\phi : \text{Bary}^N(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)) \to \mathcal{K}$ such that $(\phi \circ \text{Bary}^N)(\text{skel}^2(\Gamma_1(\sigma) \times \Gamma_2(\sigma))) \subseteq \Gamma(\sigma)$.

Operationally, the participating processes first execute protocols for T_1 and T_2, ending up on a simplex of $\mathcal{K}_1 \times \mathcal{K}_2$. More precisely, because there are at most three participants, they end up on a simplex of $\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)$. They then exchange results via N rounds of reading and writing to "scratchpad" read-write memory, ending up on a simplex of $\text{Bary}^N(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2))$. Finally, each process calls a decision map ϕ to choose a vertex in $\mathcal{K}$.

4.2 Relative Power

In this section we use the following notation for a continuous function mapping one triangle loop to another: if $\mathcal{K}_1$ and $\mathcal{K}_2$ are complexes with triangle loops $\lambda_1 = (v_0, v_1, v_2, p_{01}, p_{12}, p_{20})$ and $\lambda_2 = (w_0, w_1, w_2, q_{01}, q_{12}, q_{20})$, respectively, then we write $f : (\mathcal{K}_1, \lambda_1) \to (\mathcal{K}_2, \lambda_2)$ to denote a continuous function $f : |\mathcal{K}_1| \to |\mathcal{K}_2|$ such that $f(v_i) = w_i$ and $f(|p_{ij}|) \subseteq |q_{ij}|$.

We now state the main theorem of the paper.

▶ **Theorem 18.** *Let $T_1 = Loop(\mathcal{K}_1, \lambda_1)$, $T_2 = Loop(\mathcal{K}_2, \lambda_2)$, and $T = Loop(\mathcal{K}, \lambda)$. Then T_1 and T_2 implement T if and only if there exists a group homomorphism $h : \pi_1(\mathcal{K}_1) \times \pi_1(\mathcal{K}_2) \to \pi_1(\mathcal{K})$ such that $h([\lambda_1], [\lambda_2]) = [\lambda]$.*

Theorem 18 describes only two loop agreement tasks implementing a third, but by finite induction, one can easily generalize this to n tasks. Its proof is broken down into two other theorems, which jointly prove Theorem 18. The first theorem is a topological characterization of two tasks implementing a third, while the second theorem is on the correspondence between continuous functions and group homomorphisms.

▶ **Theorem 19.** *Tasks T_1 and T_2 implement T if and only if there exists a continuous function $f : (skel^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2) \to (\mathcal{K}, \lambda)$.*

We prove Theorem 19 by proving each direction individually via the following lemmas.

▶ **Lemma 20.** *If there is a continuous function $f : (skel^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2) \to (\mathcal{K}, \lambda)$, then T_1 and T_2 implement T.*

Proof. Suppose such a function f exists, and let Γ_1, Γ_2, and Γ be the specification maps for T_1, T_2, and T, respectively. To prove T_1 and T_2 implement T, we require an $N \in \mathbb{N}$ and a simplicial map $\phi : \text{Bary}^N(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)) \to \mathcal{K}$ such that for each $\sigma \in \mathcal{I}$, we have $(\phi \circ \text{Bary}^N)(\text{skel}^2(\Gamma_1(\sigma) \times \Gamma_2(\sigma))) \subseteq \Gamma(\sigma)$. We will construct such a ϕ by taking a simplicial approximation of a suitably defined continuous function.

Let p_{01}, p_{12}, and p_{20}, and q_{01}, q_{12}, and q_{20} be the designated edge paths of λ_1 and λ_2, respectively. Consider $X = |(p_{01} \times q_{01})| \cup |(p_{12} \times q_{12})| \cup |(p_{20} \times q_{20})| \subseteq |\mathcal{K}_1 \times \mathcal{K}_2|$ as a topological subspace. Clearly, each $|p_{ij} \times q_{ij}|$ deformation retracts to the corresponding path $|p_{ij} \star q_{ij}|$ in

$|\lambda_1 \star \lambda_2|$. In other words, we have a continuous function $H : X \times [0,1] \to |\mathcal{K}_1 \times \mathcal{K}_2|$ such that $H(x,0) = x$, $H(X,1) = |\lambda_1 \star \lambda_2|$, and $H(a,t) = a$ for each $a \in |\lambda_1 \star \lambda_2|$, $x \in X$, and $t \in [0,1]$. Now using Fact 11, we can extend H to a continuous function $H' : |\mathcal{K}_1 \times \mathcal{K}_2| \times [0,1] \to |\mathcal{K}_1 \times \mathcal{K}_2|$. In particular, define $r : |\mathcal{K}_1 \times \mathcal{K}_2| \to |\mathcal{K}_1 \times \mathcal{K}_2|$ as $r(x) = H(x,1)$. This is a continuous function from $|\mathcal{K}_1 \times \mathcal{K}_2|$ to itself that fixes $|\lambda_1 \star \lambda_2|$ while collapsing X to $|\lambda_1 \star \lambda_2|$. We restrict r to $|\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)|$ and invoke Fact 10 to get a function $g : |\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)| \to |\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)|$ that fixes $|\lambda_1 \star \lambda_2|$ while collapsing $\mathrm{skel}^2(X)$ to $|\lambda_1 \star \lambda_2|$. Now let $F = f \circ g$. This is a continuous function $F : |\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)| \to |\mathcal{K}|$ which maps $\lambda_1 \star \lambda_2$ to λ.

To show F is carried by Γ, first consider the case where $|\sigma| = 1$. Then the point $|\Gamma_1(\sigma) \times \Gamma_2(\sigma)|$ is contained in $|\lambda_1 \star \lambda_2|$, so is fixed under g, and hence mapped to the appropriate point in λ by the given function f. The case $|\sigma| = 2$ is similar. We have $|\Gamma_1(\sigma) \times \Gamma_2(\sigma)| \subseteq X$, which collapses to $|\lambda_1 \star \lambda_2|$ under g. The function f maps this to λ, as desired. The final case is when $|\sigma| = 3$, which does not require any part of the proof above, since $\Gamma(\sigma) = \mathcal{K}$. In all cases, we see that F is carried by Γ. Letting $\phi : \mathrm{Bary}^N(\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)) \to \mathcal{K}$ be a simplicial approximation of F, ϕ is also carried by Γ, so we have the required decision map. ◀

▶ **Lemma 21.** *If tasks T_1 and T_2 implement T, then there is a continuous function $f : (\mathit{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2) \to (\mathcal{K}, \lambda)$.*

Proof. Assuming T_1 and T_2 implement T, we have a simplicial map $\phi : \mathrm{Bary}^N(\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)) \to \mathcal{K}$ that is carried by Γ. In particular, ϕ maps $\lambda_1 \star \lambda_2$ to λ. Let $f : (\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2) \to (\mathcal{K}, \lambda)$, defined by $f(x) = |\phi|(x)$. Then f maps $|\lambda_1 \star \lambda_2|$ to $|\lambda|$ since ϕ does this as well. ◀

Lemmas 20 and 21 together prove Theorem 19. Next, we prove the correspondence between continuous functions and group homomorphisms. In order to do this, we refer to the following result shown in Herlihy and Rajsbaum [9].

▶ **Lemma 22.** *Let $\mathcal{K}$ and $\mathcal{L}$ be finite, connected, 2-dimensional simplicial complexes, and let $h : \pi_1(\mathcal{K}) \to \pi_1(\mathcal{L})$ be a homomorphism with $h([\sigma]) = [\tau]$. Then there exists a continuous $f : |\mathcal{K}| \to |\mathcal{L}|$ such that $f_* = h$ and $f \circ \sigma = \tau$.*

▶ **Theorem 23.** *There exists a continuous function $f : (\mathit{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2) \to (\mathcal{K}, \lambda)$ if and only if there exists a group homomorphism $h : \pi_1(\mathcal{K}_1) \times \pi_1(\mathcal{K}_2) \to \pi_1(\mathcal{K})$ such that $h([\lambda_1], [\lambda_2]) = [\lambda]$.*

Proof. First suppose we have a continuous function $f : (\mathrm{skel}^2(|\mathcal{K}_1 \times \mathcal{K}_2|), \lambda_1 \star \lambda_2) \to (\mathcal{K}, \lambda)$. We begin by constructing a homomorphism $h' : \pi_1(|\mathcal{K}_1 \times \mathcal{K}_2|) \to \pi_1(\mathcal{K})$ with $h'([\lambda_1 \star \lambda_2]) = [\lambda]$. Let $\iota : \mathrm{skel}^2(|\mathcal{K}_1 \times \mathcal{K}_2|) \to |\mathcal{K}_1 \times \mathcal{K}_2|$ be the inclusion map, whose induced homomorphism is actually an isomorphism, by Fact 9. Then we let $h' = f_* \circ \iota_*^{-1}$. In order to show $h'([\lambda_1 \star \lambda_2]) = [\lambda]$, it suffices to show that $\iota_*^{-1}([\lambda_1 \star \lambda_2]) = [\lambda_1 \star \lambda_2]$. However, notice that $[\lambda_1 \star \lambda_2] = \iota_*([\lambda_1 \star \lambda_2])$ since $\lambda_1 \star \lambda_2$ is already in $\mathrm{skel}^2(|\mathcal{K}_1 \times \mathcal{K}_2|)$, so $\iota_*^{-1}([\lambda_1 \star \lambda_2]) = [\lambda_1 \star \lambda_2]$ as required.

Now, we define the desired homomorphism $h : \pi_1(\mathcal{K}_1) \times \pi_1(\mathcal{K}_2) \to \pi_1(\mathcal{K})$ using h'. Let α_1 and α_2 be loops in $\mathcal{K}_1$ and $\mathcal{K}_2$ respectively. By Fact 10, α_1 and α_2 are homotopic to edge loops β_1 and β_2. Now define h as $h([\alpha_1], [\alpha_2]) = h'([\beta_1 \star \beta_2])$. Then it follows that $h([\lambda_1], [\lambda_2]) = [\lambda]$. To show h' is well-defined, we need to show that $|\beta_1 \star \beta_2| \simeq |\beta_1' \star \beta_2'|$ for other edge-loop representatives β_1' and β_2' of α_1 and α_2. We can find edge homotopies H_1 and H_2 taking β_1 and β_2 to β_1' and β_2', respectively, so $H_1 \star H_2$ is an edge homotopy from

$|\beta_1 \star \beta_2| \simeq |\beta_1' \star \beta_2'|$, proving that h is well-defined. We have thus found the required h, which proves the forward direction of the theorem.

Now suppose we start with a homomorphism h as described above. We reverse the above argument. We begin by constructing a homomorphism $h' : \pi_1(|\mathcal{K}_1 \times \mathcal{K}_2|) \to \pi_1(\mathcal{K})$. Let α be a loop in $|\mathcal{K}_1 \times \mathcal{K}_2|$. As before, α is homotopic to some edge loop β of $\mathcal{K}_1 \times \mathcal{K}_2$. We define $h'([\alpha]) = h([\rho_1 \circ \beta], [\rho_2 \circ \beta])$, where the ρ_i are the projection maps. This map is clearly well-defined and a homomorphism since it is the composition of h and the induced maps of the ρ_i.

Now we define a homomorphism $h'' : \pi_1(\text{skel}^2(|\mathcal{K}_1 \times \mathcal{K}_2|)) \to \pi_1(\mathcal{K})$ with $h''([\lambda_1 \star \lambda_2]) = [\lambda]$, using h'. Let ι be the inclusion map, as before. Then we define $h'' = h' \circ \iota_*$. Since $\iota_*([\lambda_1 \star \lambda_2]) = [\lambda_1 \star \lambda_2]$, we see that $h''([\lambda_1 \star \lambda_2]) = [\lambda]$. Finally, we invoke Lemma 22 on h'' to obtain the required f. This proves the backward direction of the theorem, and completes the proof. ◀

Theorems 19 and 23 together prove Theorem 18.

4.3 Composite Loop Agreement

In defining multiple implementation, we said that tasks T_1 and T_2 implement T if we can use the combined output complex $\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)$ of T_1 and T_2 to solve T. We can think of parallel execution of protocols for T_1 and T_2 as solving a task with input complex Δ^2, output complex $\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)$, and specification $\Gamma_1 \times \Gamma_2$. We get a task $T' = (\Delta^2, \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \Gamma_1 \times \Gamma_2)$, and from the definitions it is clear that T_1 and T_2 implement T if and only if T' implements T. Unfortunately, T' is not a loop agreement task, since processes starting on an edge in Δ^2 can land on any edge in $\lambda_1 \times \lambda_2$ and still obey the task specification. However, the subcomplex $\lambda_1 \times \lambda_2$ is not a loop. We address this by defining a loop agreement task $T_1 \times T_2$ with output complex $\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)$ with triangle loop $\lambda_1 \star \lambda_2$. We then show that T' and $T_1 \times T_2$ implement one another, so are equivalent.

▶ **Definition 24.** Let $T_1 = \text{Loop}(\mathcal{K}_1, \lambda_1)$ and $T_2 = \text{Loop}(\mathcal{K}_2, \lambda_2)$ be loop agreement tasks. Then the *composition* of T_1 and T_2, denoted $T_1 \times T_2$, is the loop agreement task $\text{Loop}(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2)$.

▶ **Proposition 25.** *Tasks T_1 and T_2 implement $T_1 \times T_2$.*

Proof. This is an immediate consequence of Lemma 20. ◀

▶ **Proposition 26.** *Task $T_1 \times T_2$ implements T_1 (respectively T_2).*

Proof. Lemma 6.2 from Herlihy and Rajsbuam [9] that it suffices to show there is a continuous function $f : \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2) \to \mathcal{K}_1$ mapping $\lambda_1 \star \lambda_2$ to λ_1. It is easy to see that the projection map $\rho_1 : \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2) \to \mathcal{K}_1$ satisfies this condition. The proof that $T_1 \times T_2$ implements T_2 is identical. ◀

5 Category Theory of Loop Agreement

In this section, we describe a more formal connection between the class of loop agreement tasks and the class of groups, using the language of category theory. We formalize the correspondence between loop agreement tasks and algebraic signatures, and also state one direction of the main theorem using category-theoretic formalism. Intuitively, loop agreement tasks form an organized collection of objects called a "category", with decision maps, or "morphisms", connecting two tasks if one implements the other. The algebraic signature assignment, an example of a "functor" between categories, transforms the loop agreement category into a category of groups. The composition of loop agreement tasks as defined in this paper is actually their "categorical" product.

We begin with some necessary background in category theory; see Mac Lane [14] for a rigorous approach.

5.1 Categories

A *category* C consists of a collection of *objects*, denoted $\mathrm{Ob}(C)$, and a collection of *morphisms* between those objects, denoted $\mathrm{Hom}(C)$. Each morphism has a *domain* and *codomain*, which are both objects in $\mathrm{Ob}(C)$. If f is a morphism with domain X and codomain Y, we write $f : X \to Y$. This notation is suggestive of set functions, which indeed form a category.

As with set functions, morphisms can be composed. Formally, $\mathrm{Hom}(C)$ is equipped with a binary operation called *composition.* If f and g are morphisms, then their composition is denoted $f \circ g$. Note that function composition is only defined when the codomain of the first morphism is equal to the domain of the second. Composition is required to be associative; that is, given $f : W \to X$, $g : X \to Y$, and $h : Y \to Z$, we must have $h \circ (g \circ f) = (h \circ g) \circ f$. Composition also requires an identity morphism for each object X, denoted id_X, such that for each $f : X \to Y$, we have $f \circ \mathrm{id}_X = f = \mathrm{id}_Y \circ f$.

Sets and set functions comprise the category of sets, denoted **Set**. The category of topological spaces, denoted **Top**, has spaces as its objects and continuous functions as its morphisms. There is also the category of groups, **Grp**, consisting of groups and groups homomorphisms. Algebraic signatures belong to a similar category called the category of *pointed* groups, **pGrp**, whose objects are groups with distinguished elements and whose morphisms are group homomorphisms that preserve distinguished elements.

We can transform objects and morphisms of one category to objects and morphisms of another. Given categories C and D, a *functor* $F : C \to D$ assigns to each object $X \in \mathrm{Ob}(C)$ an object $F(X) \in \mathrm{Ob}(D)$, and to each morphism $f : X \to Y$ a morphism $F(f) : F(X) \to F(Y)$. Functors must respect composition; that is, given two compatible morphisms $f, g \in \mathrm{Hom}(C)$, we must have $F(f \circ g) = F(f) \circ F(g)$. Functors must also respect identity morphisms: $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$. A common example of a functor is the fundamental group functor $\pi_1 : \mathbf{pTop} \to \mathbf{Grp}$, which maps pointed topological spaces to their respective fundamental groups, and maps continuous functions to their induced homomorphisms. The geometric realization $|\cdot| : \mathbf{SimC} \to \mathbf{Top}$ is a functor from the category of simplicial complexes with simplicial maps to **Top**, which maps complexes and simplicial maps to their respective geometric realizations.

We can also combine two objects from a category to produce a new one, which is an operation called the categorical product. The categorical product of two objects is the most general object that maps onto the original two.

▶ **Definition 27.** Let C be a category, and let X_1 and X_2 be objects in this category. The *categorical product* of X_1 and X_2 is the unique object $X_1 \times X_2$ satisfying the following: there exist morphisms (called *projections*) $\rho_1 : X_1 \times X_2 \to X_1$ and $\rho_2 : X_1 \times X_2 \to X_2$ such that for any object X with morphisms $f_1 : X \to X_1$ and $f_2 : X \to X_2$, there exists a unique morphism $f : X \to X_1 \times X_2$ such that $f_1 = \rho_1 \circ f$ and $f_2 = \rho_2 \circ f$. That is, f_1 and f_2 factor through $X_1 \times X_2$ in a unique way, via f. The morphism f is called the *product morphism* of f_1 and f_2.

5.2 The Category of Loop Agreement Tasks

We define **Loop**, the category of loop agreement tasks. We let Ob(**Loop**) be the collection of all loop agreement tasks $\text{Loop}(\mathcal{K}, \lambda)$, where $\mathcal{K}$ ranges over all finite connected 2-dimensional complexes and λ ranges over all edge loops. Morphisms in **Loop** are valid decision maps between tasks. That is, given tasks $T_1 = \text{Loop}(\mathcal{K}_1, \lambda_1)$ and $T_2 = \text{Loop}(\mathcal{K}_2, \lambda_2)$, a morphism $f : T_1 \to T_2$ is a pair (δ, N) where $N \in \mathbb{N}$ and $\delta : \text{Bary}^N(\mathcal{K}_1) \to \mathcal{K}_2$ is a decision map such that T_1 solves T_2 via δ. Composition of morphisms is defined as follows. Given objects $T_1 = \text{Loop}(\mathcal{K}_1, \lambda_1)$, $T_2 = \text{Loop}(\mathcal{K}_2, \lambda_2)$, $T_3 = \text{Loop}(\mathcal{K}_3, \lambda_3)$, and morphisms $f_1 : T_1 \to T_2$, $f_2 : T_2 \to T_3$ where $f_1 = (\delta_1, N_1)$ and $f_2 = (\delta_2, N_2)$, the composition $f_2 \circ f_1$ is defined[3] as $(\delta_2 \circ \text{Bary}^{N_2}(\delta_1), N_1 + N_2)$. Two morphisms are considered equivalent if their simplicial maps are homotopic[4]. We must now prove that **Loop** is a category.

▶ **Theorem 28.** **Loop** *is a category.*

Proof. Let T_i and f_i be defined as above, and let Γ_i be the tasks' respective specification maps. To show **Loop** is a category, we need to show that Hom(**Loop**) is closed under composition, composition is associative, and identity morphisms exist. Showing that Hom(**Loop**) is closed under composition amounts to showing that T_1 solves T_3 via $\delta_2 \circ \text{Bary}^{N_2}(\delta_1) : \text{Bary}^{N_1+N_2}(\mathcal{K}_1) \to \mathcal{K}_3$. For brevity we define $\delta = \delta_2 \circ \text{Bary}^{N_2}(\delta_1)$.

From the definition of task implementation, we know that $\delta_1 \circ \text{Bary}^{N_1} \circ \Gamma_1 \subseteq \Gamma_2$ and $\delta_2 \circ \text{Bary}^{N_2} \circ \Gamma_2 \subseteq \Gamma_3$, and we want to show $\delta \circ \text{Bary}^{N_1+N_2} \circ \Gamma_1 \subseteq \Gamma_3$. So $\delta_2 \circ \text{Bary}^{N_2} \circ \delta_1 \circ \text{Bary}^{N_1} \circ \Gamma_1 \subseteq \delta_2 \circ \text{Bary}^{N_2} \circ \Gamma_2 \subseteq \Gamma_3$. We know that $\text{Bary}^{N_2} \circ \delta_1 = \text{Bary}^{N_2}(\delta_1) \circ \text{Bary}^{N_2}$, so $\delta_2 \circ \text{Bary}^{N_2} \circ \delta_1 \circ \text{Bary}^{N_1} \circ \Gamma_1 = \delta_2 \circ \text{Bary}^{N_2}(\delta_1) \circ \text{Bary}^{N_2} \circ \text{Bary}^{N_1} \circ \Gamma_1 = \delta \circ \text{Bary}^{N_1+N_2} \circ \Gamma_1 \subseteq \Gamma_3$. Therefore T_1 solves T_3 via δ, so Hom(**Loop**) is closed under our definition of composition.

Verifying associativity follows a similar argument. Again, let T_i and f_i be defined as above, and in addition let $T_4 = \text{Loop}(\mathcal{K}_4, \lambda_4)$ and let $f_3 : T_3 \to T_4$ with $f_3 = (\delta_3, N_3)$. We must show that $(f_3 \circ f_2) \circ f_1 = f_3 \circ (f_2 \circ f_1)$. But $(f_3 \circ f_2) \circ f_1 = (\delta_3 \circ \text{Bary}^{N_3}(\delta_2), N_2 + N_3) \circ (\delta_1, N_1) = (\delta_3 \circ \text{Bary}^{N_3}(\delta_2) \circ \text{Bary}^{N_2+N_3}(\delta_1), N_1 + N_2 + N_3)$, and $f_3 \circ (f_2 \circ f_1) = (\delta_3, N_3) \circ (\delta_2 \circ \text{Bary}^{N_2}(\delta_1), N_1 + N_2) = (\delta_3 \circ \text{Bary}^{N_3}(\delta_2 \circ \text{Bary}^{N_2}(\delta_1)), N_1 + N_2 + N_3) = (\delta_3 \circ \text{Bary}^{N_3}(\delta_2) \circ \text{Bary}^{N_2+N_3}(\delta_1), N_1 + N_2 + N_3)$, so $(f_3 \circ f_2) \circ f_1 = f_3 \circ (f_2 \circ f_1)$. Therefore composition is associative.

The last requirement, existence of identity morphisms, is trivial to show. Task T_1 solves itself via the decision map $(\text{id}_{\mathcal{K}_1}, 0)$. This finishes the proof that **Loop** is a category. ◀

Next, we show that the algebraic signature of Herlihy and Rajsbaum can be formulated as a functor between **Loop** and **pGrp**.

▶ **Definition 29.** Let $T_1, T_2 \in \text{Ob}(\textbf{Loop})$ with $T_1 = \text{Loop}(\mathcal{K}_1, \lambda_1)$ and $T_2 = \text{Loop}(\mathcal{K}_2, \lambda_2)$, and let $f_1 : T_1 \to T_2$ with $f_1 = (\delta_1, N_1)$ be a morphism between the two. Then the *algebraic signature functor* is a functor $S : \textbf{Loop} \to \textbf{pGrp}$ defined as follows. Object T_1 is mapped to $(\pi_1(\mathcal{K}_1), [\lambda_1])$, while morphism $f_1 : T_1 \to T_2$ is mapped to $|\delta_1|_* : (\pi_1(\mathcal{K}_1), [\lambda_1]) \to (\pi_2(\mathcal{K}_2), [\lambda_2])$.

▶ **Theorem 30.** $S : \textbf{Loop} \to \textbf{pGrp}$ *is a functor.*

[3] If $\phi : \mathcal{A} \to \mathcal{B}$ is a simplicial map, then we can define the map $\text{Bary}(\phi) : \text{Bary}(\mathcal{A}) \to \text{Bary}(\mathcal{B})$ as one that maps barycenters to barycenters. It is easy to verify that $\text{Bary}(\phi)$ is simplicial.

[4] By identifying morphisms (in this case homotopic ones), we are constructing a *quotient category* of the original one. In order to construct a quotient category, the equivalence must be compatible with composition. However, we know that homotopy is compatible with compositions of continuous functions.

Proof. We use the fact that π_1 and $|\cdot|$ are both functors. We need to show that S preserves identity morphisms and respects composition of morphisms. Let T_1, T_2, and f be defined as above, and let $T_3 = \text{Loop}(\mathcal{K}_3, \lambda_3)$ and let $f_2 : T_2 \to T_3$ with $f_2 = (\delta_2, N_2)$. Then, using the functoriality of π_1 and $|\cdot|$, we have $S(f_2 \circ f_1) = S((\delta_2 \circ \text{Bary}^{N_1}(\delta_1), N_1 + N_2)) = |\delta_2 \circ \text{Bary}^{N_1}(\delta_1)|_* = (|\delta_2| \circ |\text{Bary}^{N_1}(\delta_1)|)_* = |\delta_2|_* \circ |\delta_1|_* = S(f_2) \circ S(f_1)$, so S respects composition. Now let id_{T_1} be the identity morphism of T_1. Then $S(\text{id}_{T_1}) = S((\text{id}_{\mathcal{K}_1}, 0)) = |\text{id}_{\mathcal{K}_1}|_* = \text{id}_{\pi_1(\mathcal{K}_1)}$, so S also preserves identity morphisms. S is well-defined since π_1 cannot distinguish between homotopic functions. We conclude that S is a functor. ◀

▶ **Lemma 31.** *If $\mathcal{K}_1$ and $\mathcal{K}_2$ are objects in $\mathbf{SimC}_2$, then skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$ is their categorical product in $\mathbf{SimC}_2$.*

Proof. We first define projection maps $\rho_1 : \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2) \to \mathcal{K}_1$ and $\rho_2 : \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2) \to \mathcal{K}_2$ as $\rho_1(v_1, v_2) = v_1$ and $\rho_2(v_1, v_2) = v_2$. That is, the ρ_i are the restrictions to the 2-skeleton of the projection maps found in Definition 3, so they are clearly simplicial.

Now suppose we have a 2-dimensional complex $\mathcal{K}$ with simplicial maps $\delta_1 : \mathcal{K} \to \mathcal{K}_1$ and $\delta_2 : \mathcal{K} \to \mathcal{K}_2$. Then we define $\delta : \mathcal{K} \to \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)$ as $\delta(v) = (\delta_1(v), \delta_2(v))$. This is the only possible set function δ that makes the diagram commute; that is, δ is the only set function such that $\delta_1 = \rho_1 \circ \delta$ and $\delta_2 = \rho_2 \circ \delta$. This proves uniqueness, but we must also show that δ is simplicial.

Let σ be a simplex in skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$. Then $\delta_1(\sigma)$ and $\delta_2(\sigma)$ are simplexes in $\mathcal{K}_1$ and $\mathcal{K}_2$, respectively. But as we have shown, $\delta_1(\sigma) = \rho_1(\delta(\sigma))$ and $\delta_2(\sigma) = \rho_2(\delta(\sigma))$, so in particular, we see that $\rho_1(\delta(\sigma))$ and $\rho_2(\delta(\sigma))$ are simplexes. Hence by Definition 3, $\delta(\sigma)$ is a simplex in $\mathcal{K}_1 \times \mathcal{K}_2$, and furthermore it is a simplex in skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$ since the dimension of σ is at most 2. So δ is a simplicial map, which proves that skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$ is the categorical product of $\mathcal{K}_1$ and $\mathcal{K}_2$ in $\mathbf{SimC}_2$. ◀

Note that Lemma 31 easily generalizes to $\mathbf{SimC}_n$ and the n-skeleton.

▶ **Theorem 32.** *Composition of loop agreement tasks is the categorical product in* **Loop**.

Proof. Let $T_1 = \text{Loop}(\mathcal{K}_1, \lambda_1)$ and $T_2 = \text{Loop}(\mathcal{K}_2, \lambda_2)$ be tasks as defined before, and let Γ_1 and Γ_2 be their specification maps, respectively. Let $\Gamma_\times$ be the specification map of $T_1 \times T_2$. We must first define decision maps from $T_1 \times T_2$ to T_1 and T_2 that make $T_1 \times T_2$ the categorical product. We know that skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$ is the categorical product of $\mathcal{K}_1$ and $\mathcal{K}_2$ in the category $\mathbf{SimC}_2$, and that the product comes with projection maps $\rho_1 : \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2) \to \mathcal{K}_1$ and $\rho_2 : \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2) \to \mathcal{K}_2$. Using these, we define maps $g_1 : T_1 \times T_2 \to T_1$ and $g_2 : T_1 \times T_2 \to T_2$ with $g_1 = (\rho_1, 0)$ and $g_2 = (\rho_2, 0)$, and we show that these maps make $T_1 \times T_2$ the categorical product of T_1 and T_2.

We showed in Proposition 26 that g_1 and g_2 solve T_1 and T_2, respectively. To prove that g_1 and g_2 are the projection maps satisfying Definition 27, we consider a task T that implements both T_1 and T_2, say via maps $f_1 = (\delta_1, N_1)$ and $f_2 = (\delta_2, N_2)$, respectively. Let $T = \text{Loop}(\mathcal{K}, \lambda)$ and let Γ be its specification map. We must find a decision map that solves $T_1 \times T_2$ from T. Without loss of generality, assume $N_1 \geq N_2$, so let $\delta_2' : \text{Bary}^{N_1}(\mathcal{K}) \to \mathcal{K}_2$ be a simplicial approximation of δ_2. Then $\delta = (\delta_1, \delta_2')$ is a map from $\text{Bary}^{N_1}(\mathcal{K})$ to skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$, though it does not necessarily carry λ to $\lambda_1 \star \lambda_2$. Instead, $g = (\delta, N_1)$ is a morphism from $\text{Loop}(\mathcal{K}, \lambda)$ to $\text{Loop}(\text{skel}^2(K_1 \times \mathcal{K}_2), \delta(\lambda))$. However, it is easy to see that $\delta(\lambda)$ is homotopic to $\lambda_1 \star \lambda_2$. Using Fact 11, we can extend this to a homotopy on all of skel$^2(\mathcal{K}_1 \times \mathcal{K}_2)$, so we obtain a continuous function $h : |\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)| \to |\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)|$. Let $\gamma : \text{Bary}^M(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)) \to \text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)$ be a simplicial approximation of h. Then notice that $g' = (\gamma, M)$ is a morphism from $\text{Loop}(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \delta(\lambda))$ to $\text{Loop}(\text{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2), \lambda_1 \star \lambda_2)$. So $f = g' \circ g$ is a morphism

$f : T \to T_1 \times T_2$. We must also show that $f = (\gamma \circ \mathrm{Bary}^M(\delta), N_1 + M)$ makes the diagram commute. Let $\delta' = \gamma \circ \mathrm{Bary}^M(\delta)$. We know that $\rho_i \circ \delta \simeq \delta_i$ by construction of δ, and it is also clear that $\delta' \simeq \delta$, by construction of δ' and γ. It follows that $\rho_i \circ \delta' \simeq \delta_i$, proving that f makes the diagram commute. Thus we have the required product morphism.

Finally, it remains to show that f is unique. Let f' be any such morphism making the diagram commute, and let δ' be its simplicial map. Then, as set maps, we know that $\delta' = (\rho_1 \circ \delta', \rho_2 \circ \delta')$. However, we are assuming that $|\rho_1 \circ \delta'| \simeq |\delta_1|$ and $|\rho_2 \circ \delta'| \simeq |\delta_2|$, so this allows us to conclude that $|\delta'| = (|\rho_1 \circ \delta'|, |\rho_2 \circ \delta'|) \simeq (|\delta_1|, |\delta_2|)$. Therefore $|\delta'| \simeq (|\delta_1|, |\delta_2|)$, which is homotopic to the map constructed in the existence proof above. So δ is unique up to homotopy, meaning that f is unique. This proves that g_1 and g_2 are satisfactory projection maps, proving that $T_1 \times T_2$ is in fact the categorical product of T_1 and T_2. ◀

The category **pGrp** also has products. We define this product below, and state without proof that it is indeed the categorical product. This follows immediately from the fact that the direct product of groups is the categorical product in **Grp** [14].

▶ **Fact 33.** *Let (G_1, g_1) and (G_2, g_2) be objects in* **pGrp**. *Then $(G_1 \times G_2, (g_1, g_2))$ is their categorical product.*

With this in mind, the following corollary is a simple consequence of Theorem 18.

▶ **Corollary 34.** *The functor $S :$* **Loop** $\to$ **pGrp** *preserves products.*

Proof. Let $T_1 = \mathrm{Loop}(\mathcal{K}_1, \lambda_1)$ and $T_2 = \mathrm{Loop}(\mathcal{K}_2, \lambda_2)$ be objects in **Loop**. Then $S(T_1) = (\pi_1(\mathcal{K}_1), [\lambda_1])$ and $S(T_2) = (\pi_1(\mathcal{K}_1), [\lambda_2])$, so $S(T_1) \times S(T_2) = (\pi_1(\mathcal{K}_1) \times \pi_2(\mathcal{K}_2), ([\lambda_1], [\lambda_2]))$. However, from the proof of Theorem 23, we see that $(\pi_1(\mathcal{K}_1) \times \pi_2(\mathcal{K}_2), ([\lambda_1], [\lambda_2])) \cong (\pi_1(\mathrm{skel}^2(\mathcal{K}_1 \times \mathcal{K}_2)), [\lambda_1 \star \lambda_2]) = S(T_1 \times T_2)$, so in fact $S(T_1 \times T_2) \cong S(T_1) \times S(T_2)$. Therefore S preserves products. ◀

6 Applications

In this section we present some simple applications of the correspondence between compositions of loop agreement tasks and the products of their algebraic signatures.

▶ **Proposition 35.** *Let T be $(3, 2)$-set agreement, and let T' be any other loop agreement task. Then $T \times T'$ and T are equivalent.*

Proof. Recall that $(3, 2)$-set agreement is the task $\mathrm{Loop}(\mathrm{skel}^1(\Delta^2), \zeta))$, where ζ is the triangle loop $(0, 1, 2, ((0, 1)), ((1, 2)), ((2, 0)))$. This triangle loop generates $\pi_1(\mathrm{skel}^1(\Delta^2))$, so $S(T) = (\pi_1(\mathrm{skel}^1(\Delta^2)), [\zeta]) \cong (\mathbb{Z}, 1)$. Let $S(T') = (G, g)$. Then by Corollary 34, $S(T \times T') = S(T) \times S(T') = (\mathbb{Z} \times G, (1, g))$. The homomorphism $\phi : \mathbb{Z} \times G \to \mathbb{Z}$ defined by projection onto the first coordinate sends $(1, g)$ to 1, and the homomorphism $\psi : \mathbb{Z} \to \mathbb{Z} \times G$ defined by $\psi(n) = (n, g)$ sends 1 to $(1, g)$. So $T \times T'$ and T implement one another, so they are equivalent. ◀

Since $(3, 2)$-set agreement was shown to be universal for loop agreement by Herlihy and Rajsbaum [9], it is operationally intuitive that composing it with any other loop agreement task should not change its relative power.

▶ **Proposition 36.** *Let T be any simplex agreement task, and let T' be any other loop agreement task. Then $T \times T'$ and T' are equivalent.*

Proof. Since the output complex if T is a subdivided simplex, it has trivial fundamental group, so $S(T) = (1, e)$. As before, let $S(T') = (G, g)$. By Corollary 34, $S(T \times T') = S(T) \times S(T') = (1 \times G, (g, e))$, which is clearly isomorphic to (G, g). So $T \times T'$ and T implement one another, so are equivalent. ◀

Herlihy and Rajsbaum also showed that simplex agreement is implemented from any loop agreement task [9], so it is also intuitively clear that composing a task with simplex agreement should not change the relative power of the original task.

▶ **Proposition 37.** *Let T be any loop agreement task. Then $T \times T$ and T are equivalent.*

Proof. Let $S(T) = (G, g)$. Then by Corollary 34, $S(T \times T) = S(T) \times S(T) = (G \times G, (g, g))$. Letting $\phi : G \to G \times G$ be the diagonal map $\phi(x) = (x, x)$, ϕ maps g to (g, g), and letting $\psi : G \times G \to G$ be projection onto a coordinate, ψ maps (g, g) to g. So $T \times T$ and T are equivalent. ◀

The above result states that composing a loop agreement task with copies of itself will not change its relative power.

7 Conclusions

It is a common technique to study a class of objects by mapping these objects into a class of simpler ones in such a way that preserves enough information about the original class of objects. This was the idea behind the fundamental group from algebraic topology, and was also the idea of the algebraic signature of Herlihy and Rajsbaum in their work on loop agreement. In this work we formalized and further extended the algebraic signature characterization by defining the composition of tasks and relating compositions of tasks to products of groups, and in doing so we partially answered the questions raised in the original paper. How much further can this characterization be extended; what more can we learn from the algebraic signature functor between loop agreement tasks and groups with distinguished elements? Does this functor have an adjunction?

The categorical techniques in this paper can be applied to general tasks. For example, tasks with decision maps form a category **Task**, with loop agreement as a subcategory. In the case of loop agreement, we are able to extract valuable information about tasks by mapping them into groups. What kind of functors may we apply to general tasks? Also in the case of loop agreement, we were able to identify parallel composition with the category product. Can parallel composition be defined for more general tasks, for instance via $\text{skel}^n(\mathcal{O}_1 \times \mathcal{O}_2)$, and what is its precise operational meaning of parallel composition for general tasks?

References

1 H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.

2 S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

3 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

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

5 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 329–338, 2006.

6 A. Hatcher. *Algebraic Topology.* Cambridge University Press, 2002.

7 M. P. Herlihy, D. N. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.

8 M. P. Herlihy and S. Rajsbaum. The decidability of distributed decision tasks. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 589–598, 1997.

9 M. P. Herlihy and S. Rajsbaum. A classification of wait-free loop agreement tasks. *Theor. Comput. Sci.*, 291(1):55–77, 2003.

10 M. P. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 111–120, 1993.

11 M. P. Herlihy and N. Shavit. A simple constructive computability theorem for wait-free computation. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 243–252, 1994.

12 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

13 D. N. Kozlov. *Combinatorial Algebraic Topology*, volume 21 of *Algorithms and computation in mathematics.* Springer, 2008.

14 S. Mac Lane. *Categories for the Working Mathematician.* Springer Verlag, 1998.

15 X. Liu, J. Pu, and J. Pan. A classification of degenerate loop agreement. In *Fifth IFIP International Conference On Theoretical Computer Science*, volume 273 of *IFIP*, pages 203–213. Springer, 2008.

16 X. Liu, Z. Xu, and J. Pan. Classifying rendezvous tasks of arbitrary dimension. *Theor. Comput. Sci.*, 410(21-23):2162–2173, 2009.

Loosely-Stabilizing Leader Election on Arbitrary Graphs in Population Protocols Without Identifiers nor Random Numbers*

Yuichi Sudo[1], Fukuhito Ooshita[2], Hirotsugu Kakugawa[3], and Toshimitsu Masuzawa[4]

1 **NTT Secure Platform Laboratories, Tokyo, Japan; and**
Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
sudo.yuichi@lab.ntt.co.jp

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

3 **Graduate School of Information Science and Technology, Osaka University, Osaka, Japan**
kakugawa@ist.osaka-u.ac.jp

4 **Graduate School of Information Science and Technology, Osaka University, Osaka, Japan**
masuzawa@ist.osaka-u.ac.jp

Abstract

In the population protocol model Angluin et al. proposed in 2004, there exists no self-stabilizing leader election protocol for complete graphs, arbitrary graphs, trees, lines, degree-bounded graphs and so on unless the protocol knows the exact number of nodes. To circumvent the impossibility, we introduced the concept of *loose-stabilization* in 2009, which relaxes the closure requirement of self-stabilization. A loosely-stabilizing protocol guarantees that starting from any initial configuration a system reaches a safe configuration, and after that, the system keeps its specification (e.g. the unique leader) not forever, but for a sufficiently long time (e.g. exponentially large time with respect to the number of nodes). Our previous works presented two loosely-stabilizing leader election protocols for arbitrary graphs; One uses agent identifiers and the other uses random numbers to elect a unique leader. In this paper, we present a loosely-stabilizing protocol that solves leader election on arbitrary graphs without agent identifiers nor random numbers. By the combination of virus-propagation and token-circulation, the proposed protocol achieves polynomial convergence time and exponential holding time without such external entities. Specifically, given upper bounds N and Δ of the number of nodes n and the maximum degree of nodes δ respectively, it reaches a safe configuration within $O(mn^3d + mN\Delta^2 \log N)$ expected steps, and keeps the unique leader for $\Omega(Ne^N)$ expected steps where m is the number of edges and d is the diameter of the graph. To measure the time complexity of the protocol, we assume the uniformly random scheduler which is widely used in the field of the population protocols.

1998 ACM Subject Classification G.2.2. Graph Theory

Keywords and phrases Loose-stabilization, Population protocols, and Leader election

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.14

* This work was supported by JSPS KAKENHI Grant Numbers 24500039, 26280022, 26330084, and 15H00816.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 14; pp. 14:1–14:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

This paper focuses on self-stabilizing leader election in the population protocol model. The *population protocol* (PP) model, which was presented by Angluin et al. [1], represents wireless sensor networks of mobile sensing devices that cannot control their movement. Two devices (say *agents*) communicate with each other and change their states only when they come sufficiently close to each other (we call this event an *interaction*). Self-stabilizing leader election (SS-LE) requires that starting from any configuration, a system (say *population*) reaches a safe-configuration in which a unique leader is elected, and after that, the population has the unique leader forever. Self-stabilizing leader election is important in the PP model because (i) many population protocols in the literature work on the assumption of the unique leader [1, 2, 3], and (ii) self-stabilization tolerates any finite number of transient faults and this property suits systems consisting of numerous cheap and unreliable nodes. (Such systems are the original motivation of the PP model.) However, there exists strict impossibility of SS-LE in the PP model: no protocol solves SS-LE for complete graphs, arbitrary graphs, trees, lines, degree-bounded graphs and so on unless the number of agents n is available to agents in advance [3].

Therefore, many studies of SS-LE took either one of the following two approaches. One approach is to accept the assumption that the exact n is available and focus on the space complexity of the protocol. Cai et al. [6] proved that n states of each agent is necessary and sufficient to solve SS-LE for a complete graph of n agents. Mizoguchi et al. [12] and Xu et al. [15] improved the space-complexity by adopting the mediated population protocol model [10] and the PP_k model [5] respectively. The other approach is to use *oracles*, a kind of failure detectors. Fischer and Jiang [8] took this approach for the first time. They introduced oracle $\Omega?$ that informs all agents whether a leader exists or not and proposed two protocols that solve SS-LE for rings and complete graphs by using $\Omega?$. Beauquier et al. [4] presented an SS-LE protocol for arbitrary graphs that uses two copies of $\Omega?$. Canepa et al. [7] proposed two SS-LE protocols that use $\Omega?$ and consume only 1 bit of each agent: one is a deterministic protocol for trees and the other is a probabilistic protocol for arbitrary graphs although the position of the leader is not static and moves among the agents.

Our previous works [13, 14] took another approach to solve SS-LE. We introduced the concept of *loose-stabilization*, which relaxes the closure requirement of self-stabilization. Specifically, starting from any initial configuration, the population must reach a safe configuration within a relatively short time; after that, the specification of the problem (the unique leader) must be kept for a sufficiently long time, though not forever. We proposed three loosely-stabilizing protocols P_{LE}, P_{ID}, and P_{RD}. Protocol P_{LE} solves leader election for complete graphs whose size is no more than given upper bound N of n. Protocol P_{ID} and P_{RD} solve leader election for arbitrary graphs using agent identifiers and random numbers respectively, given N and upper bound Δ of the maximum degree of nodes δ. All the three protocols are practically equivalent to a SS-LE protocol since they keep the specification for an exponentially long time after reaching a safe configuration (and reaches a safe configuration within polynomial time).

Some works on population protocols assume the probabilistic distribution regarding the interactions of agents: any interaction occurs uniformly at random [1, 2, 9, 13, 14]. This assumption have been used mainly for evaluating the time complexity of protocols. We also adopt this assumption because the measure of time is crucial in the concept of loose-stabilization. The impossibility result for SS-LE [1] still holds even with this assumption.

Table 1 Loosely-stabilizing leader election for arbitrary graphs.

Protocol	Convergence Time	Holding Time	Agent Memory	Requisite
P_{ID} [14]	$O(mN\Delta \log N)$	$\Omega(Ne^N)$	$O(\log N)$	agent identifiers
P_{RD} [14]	$O(mN^3\Delta^2 \log N)$	$\Omega(Ne^N)$	$O(\log N)$	random numbers
P_{AR} (proposed)	$O(mn^3 d + mN\Delta^2 \log N)$	$\Omega(Ne^N)$	$O(\log N)$	–

Our Contribution

This paper proposes a loosely-stabilizing protocol P_{AR} for leader election in arbitrary graphs without agent identifiers nor random numbers (or a model with a weaker assumption than P_{ID} or P_{RD}). Thus, we succeed to remove the assumptions of unique identifiers and random number generators for a loosely-stabilizing leader election on arbitrary graphs in the PP model, which may be difficult to realize in weak computation models, like the PP model, consisting of huge number of tiny devices with restricted capability.

The expected convergence time and the expected holding time of P_{ID}, P_{RD}, and P_{AR} are shown in Table 1 where d is the diameter of the graph. All the protocols including P_{AR} keep the unique leader for an exponentially long time ($\Omega(Ne^N)$ interactions) after a safe configuration. Protocol P_{AR} consumes $O(\log N)$ bits of each agent's memory while any self-stabilizing protocol (which uses knowledge of exact n) consumes $\Omega(\log n)$ memory [6]. Furthermore, Izumi [9] proves that loosely-stabilizing leader election with polynomial convergence time and exponentially long holding time needs $\Omega(\log n)$ agent memory. Thus, P_{AR} is asymptotically space-optimal when N is polynomial in n. One may think that the model of anonymous agents and $O(\log N)$ agent memory is not well-motivated because $O(\log n)$ memory is sufficient to store an identifier. However, we believe that anonymity is still an important assumption: assigning distinct identifiers to a huge number of agents is not an easy task, and memory corruption may cause conflicts of identifiers of different agents. Actually, many works assume anonymity and agent memory space of $O(\log n)$ or more (e.g. [3, 6, 12, 13, 14, 15]). In this paper, we analyze time complexities for undirected graphs for simplicity, however, it works on any *directed* graphs without modifications.

While protocol P_{AR} is based on the virus war mechanism developed for P_{RD} [14], the key idea of P_{AR} is quite novel and has a considerable contribution: The token with a count-down timer circulates in the graph, and a leader creates and spreads a black or white virus when encountering the token with zero timer value. The idea of circulating tokens and the colors of viruses are newly introduced to remove the assumption of random number generators. This technique may be useful also for other problems and/or other models.

The formal analysis of the convergence time and the holding time is another main contribution of this paper, since analyzing such complexities of loosely-stabilizing protocols is a challenging task. In particular, we analyze in the expected time until two tokens performing random walks meet in the PP model. The analysis can be applied with slight modification to estimate the expected time until a token performing random walks visits all nodes. We believe that the analysis techniques are of significant importance because existing analysis for usual random walks cannot be applied to the population protocol model: the token always moves through an edge at each step in usual random walks while, in the population protocol model, the token moves at each step with a probability depending on the degree of the node the token currently exists on. Thus, the techniques we developed open up a new path to analysis of loosely-stabilizing protocols in the PP model.

Angluin et al. [1] proves that for any population protocol P working on complete graphs, there exists a protocol that simulates P on any arbitrary graph. One may think that this simulator can translate our previous loosely-stabilizing algorithm for complete graphs [13] to a loosely-stabilizing algorithm that works for arbitrary graphs. However, it cannot work since, in this simulation, two agents swap their states when they have interactions. This swap is needed to simulate interactions between distant agents in an arbitrary graph, but it results in the execution where an elected leader moves among the population, which does not satisfy the specification of the leader election.

2 Preliminaries

This section defines the model we consider for this paper.

A *population* is a simple and weakly-connected directed graph $G(V, E)$ where V ($|V| \geq 2$) is a set of *agents* and $E \subseteq V \times V$ is a set of directed edges. Each edge represents a possible *interactions* (or communication between two agents): If $(u, v) \in E$, agents u and v can interact with each other where u serves as an *initiator* and v serves as a *responder*. We say that G is undirected if it satisfies $(u, v) \in E \Leftrightarrow (v, u) \in E$. We define $n = |V|$ and $m = |E|$.

A *protocol* $P(Q, Y, T, O)$ consists of a finite set Q of states, a finite set Y of output symbols, transition function $T : Q \times Q \to Q \times Q$, and output function $O : Q \to Y$. When an interaction between two agents occurs, T determines the next states of the two agents based on their current states. The *output of an agent* is determined by O: the output of agent v with state $q \in Q$ is $O(q)$.

A *configuration* is a mapping $C : V \to Q$ that specifies the states of all the agents. We denote the set of all configurations of protocol P by $\mathcal{C}_{\mathrm{all}}(P)$. We say that configuration C changes to C' by interaction $e = (u, v)$, denoted by $C \overset{e}{\to} C'$, if we have $(C'(u), C'(v)) = T(C(u), C(v))$ and $C'(w) = C(w)$ for all $w \in V \setminus \{u, v\}$. A scheduler determines which interaction occurs at each time. In this paper, we consider a *uniformly random scheduler* $\Gamma = \Gamma_0, \Gamma_1, \dots$: each $\Gamma_t \in E$ is a random variable such that $\Pr(\Gamma_t = (u, v)) = 1/m$ for any $t \geq 0$ and any $(u, v) \in E$. Given an initial configuration C_0 and Γ, the *execution* of protocol P is defined as $\Xi_P(C_0, \Gamma) = C_0, C_1, \dots$ such that $C_t \overset{\Gamma_t}{\to} C_{t+1}$ for all $t \geq 0$. We denote $\Xi_P(C_0, \Gamma)$ simply by $\Xi_P(C_0)$ when no confusion occurs.

The leader election problem requires that every agent should output L or F which means "leader" or "follower" respectively. We say that a finite or infinite sequence of configurations $\xi = C_0, C_1, \dots$ preserves a unique leader, denoted by $\xi \in LE$, if there exists $v \in V$ such that $O(C_t(v)) = L$ and $O(C_t(u)) = F$ for any $t \geq 0$ and $u \in V \setminus \{v\}$. For $\xi = C_0, C_1, \dots$, the holding time of the leader $\mathrm{HT}(\xi, LE)$ is defined as the maximum $t \in \mathbb{N}$ that satisfies $(C_0, C_1, \dots, C_{t-1}) \in LE$. We define $\mathrm{HT}(\xi, LE) = 0$ if $C_0 \notin LE$. We denote $\mathbf{E}[\mathrm{HT}(\Xi_P(C), LE)]$ by $\mathrm{EHT}_P(C, LE)$. Intuitively, $\mathrm{EHT}_P(C, LE)$ is the expected number of interactions for which the population keeps the unique leader after protocol P starts from configuration C. For configuration sequence $\xi = C_0, C_1, \dots$ and a set of configurations $\mathcal{C}$, we define convergence time $\mathrm{CT}(\xi, \mathcal{C})$ as the minimum $t \in \mathbb{N}$ that satisfies $C_t \in \mathcal{C}$. We define $\mathrm{CT}(\xi, \mathcal{C}) = |\xi|$ if $C_t \notin \mathcal{C}$ for any $t \geq 0$, where $|\xi|$ is the length of ξ (i.e. the number of configurations). We denote $\mathbf{E}[\mathrm{CT}(\Xi_P(C), \mathcal{C})]$ by $\mathrm{ECT}_P(C, \mathcal{C})$. Intuitively, $\mathrm{ECT}_P(C, \mathcal{C})$ is the expected number of interactions by which the population reaches a configuration in $\mathcal{C}$ when starting from C.

▶ **Definition 1** (Loose-stabilizing leader election [13]). Protocol $P(Q, Y, T, O)$ is an (α, β)-loosely-stabilizing leader election protocol if there exists set $\mathcal{S}$ of configurations satisfying $\max_{C \in \mathcal{C}_{\mathrm{all}}(P)} \mathrm{ECT}_P(C, \mathcal{S}) \leq \alpha$ and $\min_{C \in \mathcal{S}} \mathrm{EHT}_P(C, LE) \geq \beta$.

Chernoff Bounds

Two variants of Chernoff bounds [11] used in several proofs of this paper are quoted below.

▶ **Lemma 2** (Eq. (4.2) in [11]). *The following inequality holds for any binomial random variable X and any κ, $0 < \kappa \le 1$:*

$$\Pr(X \ge (1+\kappa)\mathbf{E}[X]) \le e^{-\kappa^2\mathbf{E}[X]/3}.$$

▶ **Lemma 3** (Eq. (4.5) in [11]). *The following inequality holds for any binomial random variable X and κ, $0 < \kappa \le 1$:*

$$\Pr(X \le (1-\kappa)\mathbf{E}[X]) \le e^{-\kappa^2\mathbf{E}[X]/2}.$$

3 Loosely-stabilizing Leader Election Protocol

This section presents loosely-stabilizing leader election protocol P_{AR} for arbitrary undirected anonymous graphs without identifiers or random numbers. Symmetry breaking is not a key issue to elect a leader in the population protocol model since random scheduler breaks the symmetry of the population. (Global-fairness breaks the symmetry in the case of deterministic scheduler.) The challenging issue is to reduce the number of leaders to one while avoiding to remove all leaders from the population. Protocol P_{AR} solves this issue without identifiers or random numbers by virus-propagation and token-circulation. A leader tries to kill other leaders by creating and propagating a virus while a circulating token controls the frequency of creating a virus so that eventually exactly one agent remains a leader (i.e. survives a virus war).

Protocol P_{AR} is described in Protocol 1. A state of an agent is described by a collection of variables, and a transition function is described by a pseudo code that updates variables of initiator x and responder y. We denote the value of variable `var` of agent $v \in V$ by v.`var`. We also denote the value of `var` in state $q \in Q$ by q.`var`. In P_{AR}, each agent has three binary variables `leader` $\in \{\top, \bot\}$, `token` $\in \{\top, \bot\}$ and `color` $\in \{\mathrm{BLACK}, \mathrm{WHITE}\}$, and four timers $\mathtt{timer_L}$, $\mathtt{timer_T}$, $\mathtt{timer_V}$ and $\mathtt{timer_E}$. The output function defines leaders based on variable `leader` : agent v is a leader if v.`leader` $= \top$, and a follower otherwise. We say that agent v has a token if v.`token` $= \top$ and v has a virus if $v.\mathtt{timer_V} > 0$. We also say that v is black if v.`color` $=$ BLACK, and v is white otherwise.

Protocol P_{AR} consists of five parts: leader-creation (Lines 1–7), token-creation (Lines 8–14), token-circulation (Lines 15–20), virus-creation (Lines 29–37), and virus-propagation (Lines 21–28). Our goal is to elect a unique leader in the population from an arbitrary initial configuration. The leader-creation part creates a leader when no leader exists in the population. The other four parts work together to reduce the number of leaders to one when two ore more leaders exist.

The leader-creation part aims to create a leader when no leader exists in the population. Each agent uses $\mathtt{timer_L}$ as the barometer for suspecting that there exists no leader. Specifically, when initiator x and responder y interact, they take the larger value of $x.\mathtt{timer_L}$ and $y.\mathtt{timer_L}$, decrease it by one, and substitute the decreased value into $x.\mathtt{timer_L}$ and $y.\mathtt{timer_L}$ (Line 1). We call this event *larger value propagation.* If x or y is a leader, both timers are reset to $t_{\max}$ (Lines 2–3). We call this event *timer reset.* When a timer becomes zero (i.e. timeout), agents x and y suspect that there exists no leader in the population. Then, x becomes a new leader with the full timer value $t_{\max}$ (Lines 5–6). When no leader exists, the population never experiences timer reset, thus, their timers keep on decreasing. Hence, the timeout eventually

Algorithm 1 Leader Election P_{AR}

```
Variables of each agent:
    $\mathtt{leader} \in \{\top, \bot\}$, $\mathtt{token} \in \{\top, \bot\}$, $\mathtt{color} \in \{\mathrm{BLACK}, \mathrm{WHITE}\}$
    $\mathtt{timer_L} \in [0, t_{\max}]$, $\mathtt{timer_T} \in [0, t_{\max}]$, $\mathtt{timer_V} \in [0, t_{\mathrm{virus}}]$, $\mathtt{timer_E} \in [0, t_{\mathrm{epi}}]$
Output function $O$:
    if $v.\mathtt{leader} = \top$ holds, then the output of agent $v$ is $L$, otherwise $F$.
Interaction between initiator $x$ and responder $y$:
  $x.\mathtt{timer_L} \leftarrow y.\mathtt{timer_L} \leftarrow \max(x.\mathtt{timer_L} - 1,\ y.\mathtt{timer_L} - 1,\ 0)$
  if $x.\mathtt{leader} = \top$ or $y.\mathtt{leader} = \top$ then
      $x.\mathtt{timer_L} \leftarrow y.\mathtt{timer_L} \leftarrow t_{\max}$            // a leader resets leader timer
  else if $x.\mathtt{timer_L} = 0$ then
      $x.\mathtt{leader} \leftarrow \top$            // a new leader is created at timeout
      $x.\mathtt{timer_L} \leftarrow y.\mathtt{timer_L} \leftarrow t_{\max}$
  end if

  $x.\mathtt{timer_T} \leftarrow y.\mathtt{timer_T} \leftarrow \max(x.\mathtt{timer_T} - 1,\ y.\mathtt{timer_T} - 1,\ 0)$
  if $x.\mathtt{token} = \top$ or $y.\mathtt{token} = \top$ then
      $x.\mathtt{timer_T} \leftarrow y.\mathtt{timer_T} \leftarrow t_{\max}$            // a token resets token timer
  else if $x.\mathtt{timer_T} = 0$ then
      $x.\mathtt{token} \leftarrow \top$            // a new token is created at timeout
      $x.\mathtt{timer_T} \leftarrow y.\mathtt{timer_T} \leftarrow t_{\max}$
  end if

  $x.\mathtt{token} \leftrightarrow y.\mathtt{token}$            // a token moves between agents
  $x.\mathtt{timer_E} \leftarrow \max(0, y.\mathtt{timer_E} - 1)$
  $y.\mathtt{timer_E} \leftarrow \max(0, x.\mathtt{timer_E} - 1)$            // decrement and swap epidemic timers
  if $x.\mathtt{token} = \top$ and $y.\mathtt{token} = \top$ then
      $y.\mathtt{token} \leftarrow \bot$
  end if

  if $x.\mathtt{timer_V} > 0$ and $y.\mathtt{timer_V} = 0$ and $x.\mathtt{color} \neq y.\mathtt{color}$ then
      $y.\mathtt{leader} \leftarrow \bot$
      $y.\mathtt{color} \leftarrow x.\mathtt{color}$
  else if $x.\mathtt{timer_V} = 0$ and $y.\mathtt{timer_V} > 0$ and $x.\mathtt{color} \neq y.\mathtt{color}$ then
      $x.\mathtt{leader} \leftarrow \bot$
      $x.\mathtt{color} \leftarrow y.\mathtt{color}$
  end if
  $x.\mathtt{timer_V} \leftarrow y.\mathtt{timer_V} \leftarrow \max(x.\mathtt{timer_V} - 1,\ y.\mathtt{timer_V} - 1,\ 0)$

  if $x.\mathtt{leader} = \top$ and $x.\mathtt{token} = \top$ and $x.\mathtt{timer_E} = 0$ then
      if $x.\mathtt{color} = \mathrm{BLACK}$ then $x.\mathtt{color} \leftarrow \mathrm{WHITE}$ else $x.\mathtt{color} \leftarrow \mathrm{BLACK}$ endif
      $x.\mathtt{timer_V} \leftarrow t_{\mathrm{virus}}$
      $x.\mathtt{timer_E} \leftarrow t_{\mathrm{epi}}$
  else if $y.\mathtt{leader} = \top$ and $y.\mathtt{token} = \top$ and $y.\mathtt{timer_E} = 0$ then
      if $y.\mathtt{color} = \mathrm{BLACK}$ then $y.\mathtt{color} \leftarrow \mathrm{WHITE}$ else $y.\mathtt{color} \leftarrow \mathrm{BLACK}$ endif
      $y.\mathtt{timer_V} \leftarrow t_{\mathrm{virus}}$
      $y.\mathtt{timer_E} \leftarrow t_{\mathrm{epi}}$
  end if
```

occurs and a leader is created. When a leader exists, the timeout rarely happens since all agents keep high timer values thanks to the timer reset and the larger value propagation. Therefore, this mechanism rarely ruins stability of the unique leader.

Protocol P_{AR} reduces the number of leaders to one as follows. The token-creation part creates a token when no token exists in the population; The token-circulation part reduces the number of tokens to one, circulates the unique token among the population, and decrements the epidemic timer ($\mathtt{timer_E}$) of the unique token every time it moves; The virus-creation part creates a new virus when a leader meets a token with epidemic timer of value zero; The virus-propagation part propagates the virus to the whole population, which changes leader agents to follower agents.

The token-creation part (Lines 8–14) creates a token in the same way as the leader-creation part when no token exists in the population. There is no difference between the two parts except that the former uses variable $\mathtt{timer_T}$ while the latter uses $\mathtt{timer_L}$.

The token-circulation part (Lines 15–20) aims to reduce the number of tokens to one, and circulates the unique token. A token moves between agents by interaction (Line 15). We can say that a token makes a random walk among the population since the scheduler randomly chooses two agents to interact at each time. Hence, two tokens eventually meet if two or more tokens exist in the population. When two agents interact and both agents have tokens, then either one of the two loses its token (Lines 18–20). Hence, the number of tokens eventually becomes one. Each token has an epidemic timer ($\mathtt{timer_E}$). The epidemic timer is decremented by one every time the token moves, and thus, it becomes zero eventually (Line 16–17). Note that the number of tokens never becomes zero once a token exists since the number of tokens decreases only when two tokens meet at an interaction.

A virus-creation part (Lines 29–37) creates a new virus when a leader meets a token with an epidemic timer of value zero. We call this event *virus creation*. Specifically, if a token with $\mathtt{timer_E} = 0$ moves to a leader agent, the leader changes its color from black to white or from white to black (Lines 30 and 34) and creates a new virus with full value TTL (Time To Live), i.e. $\mathtt{timer_V} = t_{\mathrm{virus}}$ (Lines 31 and 35). The leader also resets the epidemic timer of the token (Lines 32 and 36), which enables periodical occurrence of epidemics.

A virus-propagation part (Lines 21–28) propagates a virus from agent to agent and reduces the number of leaders. When an agent has a virus (i.e. $v.\mathtt{timer_V} > 0$), we regard that $v.\mathtt{timer_V}$ is the TTL of the virus. A virus vanishes from the agent when its TTL becomes zero. In the same way as $\mathtt{timer_L}$ and $\mathtt{timer_T}$, a virus propagates at interaction in the larger value propagation fashion (Line 28). Moreover, a virus has the power to change the colors of agents and kill leaders. Specifically, if an agent with a virus interacts an agent without a virus, the virus changes the color of the newly infected agent (Lines 23 and 26). At this time, if the newly infected agent is a leader, the virus kills the leader (i.e. changes the newly infected agent from a leader to a follower). Once a new virus is created at the virus-creation part, the virus propagates to the whole population within a short time. However, the value of $\mathtt{timer_V}$ is reset only when a new virus is created. Hence, viruses eventually vanish from the population if the frequency of epidemics, controlled by the value t_{epi}, is sufficiently low. The concept of colors helps to avoid the suicide of leaders, i.e. a leader is rarely killed by a virus that it creates. Consider that a white leader creates a virus. After that, the leader and any infected agent with the virus are black, thus the leader is never killed by the virus until another virus is created and the leader becomes white.

Protocol P_{AR} correctly works if $t_{\max}$ and t_{virus} is sufficiently large and t_{epi} is sufficiently greater than t_{virus}. When there exists no leader, the leader-creation part eventually creates a leader by timeout. In the following, let us consider the case that multiple leaders exist in the population, and see how P_{AR} reduces these leaders to one. The token-creation and the token circulation parts eventually create the unique token and circulate it in the population. Since t_{epi} is sufficiently greater than t_{virus}, the population eventually reaches a configuration where no agent has virus. After that, the epidemic timer of the token keeps on decreasing and eventually becomes zero, and the token eventually moves to a leader in the population, which creates a new virus. This virus soon propagates among the whole population and turn all the agents to the ones with the same color (black or white). Let the color be black without loss of generality. Again, the virus vanishes, the epidemic timer of the token becomes zero, and the token moves to a leader in the same way. Then, the black leader becomes white and creates a new virus. It soon propagates to the whole population and changes all agents from black to white, which kills all other leaders. Then, we have the exactly one leader in the population.

Even after we have exactly one leader and one token, the population sometimes enters the wrong configuration where no leader exists, multiple leaders exist, or multiple tokens exist. These deviations are caused by the following events: (i) leader timeout happens, (ii) token timeout happens, or (iii) a new virus is created when viruses remain in the population. Cases (i) and (ii) rarely happens thanks to the timer reset, the larger value propagation, and the sufficiently large t_{max}, which is the reset value of leader timers and token timers. Case (iii) also rarely happens because t_{epi}, the reset value of the epidemic timer, is sufficiently larger than the reset value of a virus timer t_{virus}. As we shall see later, the expected time from a safe configuration to such a wrong configuration is exponential.

4 Complexity Analysis

This section analyzes the expected holding time and the expected convergence time of P_{AR}. Due to the lack of space, we present only proof sketches for the analyses of the expected convergence time. Complete proofs are left to the full paper. Notations and assumptions used in this paper are summarized in Table 2.

We have three parameters in P_{AR}: the reset values of timers t_{max}, t_{virus}, and t_{epi}. We mentioned that P_{AR} correctly works if t_{max} and t_{virus} is sufficiently large and t_{epi} is sufficiently greater than t_{virus}. Specifically, we assume $t_{\mathrm{max}} \geq 8\delta \max(d, \lceil 2 \log mn^3 d \rceil)$, $t_{\mathrm{virus}} = t_{\mathrm{max}}/2$, and $t_{\mathrm{epi}} \geq 4\delta t_{\mathrm{max}} \lceil \log n \rceil$ where δ is the maximum degree of the agents and d is the diameter of population G. (Note that δ is an even number because G is undirected, i.e. $(u, v) \in E \Leftrightarrow (v, u) \in E$.) We also assume that t_{epi} is not extremely large: $t_{\mathrm{epi}} \leq \tau e^{\tau}/(9n)$ where $\tau = \lfloor t_{\mathrm{max}}/(8\delta) \rfloor$. Otherwise, even if a leader exists, the leader timeout happens with non-negligible probability within an exponentially long epidemic interval. This means that the protocol may not reduce the number of leaders to one at the convergence step. We also assume $n \geq 3$ because P_{AR} is obviously a self-stabilizing leader election protocol when $n = 2$. In the rest of this section, we prove the following equations under these assumptions:

$$\max_{C \in \mathcal{C}_{\mathrm{all}}} \mathrm{ECT}_{P_{\mathrm{AR}}}(C, \mathcal{S}_{\mathrm{AR}}) = O(mn^3 d + m t_{\mathrm{epi}}), \tag{1}$$

$$\min_{C \in \mathcal{S}_{\mathrm{AR}}} \mathrm{EHT}_{P_{\mathrm{AR}}}(C, LE) = \Omega(\tau e^{\tau}), \tag{2}$$

where $\mathcal{S}_{\mathrm{AR}}$ is the set of configurations we define later. When upper bounds N and Δ of n and δ are available and we assign $t_{\mathrm{max}} = 8\Delta \max(N, \lceil 12 \log N \rceil)$, $t_{\mathrm{epi}} = 4\Delta t_{\mathrm{max}} \lceil \log N \rceil$, then P_{AR} is an $(O(mn^3 d + mN\Delta^2 \log N), \Omega(Ne^N))$-loosely-stabilizing leader election protocol. (Note that this assignment satisfies the above assumptions.)

Table 2 Notations and Assumptions for P_{AR}.

Notations	
τ :	$\lfloor t_{\max}/(8\delta) \rfloor$
$\#_L(C)$:	the number of leaders in configuration C
$\#_T(C)$:	the number of tokens in configuration C
$\mathcal{L}_{\mathrm{one}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_L(C) = 1\}$
$\mathcal{T}_{\mathrm{one}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_T(C) = 1\}$
$\mathcal{L}_{\mathrm{exist}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_L(C) \geq 1\}$
$\mathcal{T}_{\mathrm{exist}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_T(C) \geq 1\}$
$\mathcal{L}_{\mathrm{half}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{timer_L} > t_{\max}/2\}$
$\mathcal{T}_{\mathrm{half}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{timer_T} > t_{\max}/2\}$
$\mathcal{V}_{\mathrm{same}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \exists u, \forall v \in V,\ C(u).\mathtt{leader} = \top$ $\wedge\ (C(v).\mathtt{timer_V} > 0 \Rightarrow C(u).\mathtt{color} = C(v).\mathtt{color})\}$
$\mathcal{V}_{\mathrm{zero}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{timer_V} = 0\}$
$\mathcal{E}_{\mathrm{half}}$:	$\{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{token} = \top \Rightarrow C(v).\mathtt{timer_E} > t_{\mathrm{epi}}/2\}$
$\mathcal{S}_{\mathrm{AR}}$:	$\mathcal{L}_{\mathrm{one}} \cap \mathcal{T}_{\mathrm{one}} \cap \mathcal{L}_{\mathrm{half}} \cap \mathcal{T}_{\mathrm{half}} \cap \mathcal{V}_{\mathrm{same}} \cap (\mathcal{E}_{\mathrm{half}} \cup \mathcal{V}_{\mathrm{zero}})$
$\mathrm{PROP}_L(i)$:	$C_{2m\tau(i+1)} \in \mathcal{L}_{\mathrm{half}} \vee C_{2m\tau i} \notin \mathcal{L}_{\mathrm{exist}}$
$\mathrm{PROP}_T(i)$:	$C_{2m\tau(i+1)} \in \mathcal{T}_{\mathrm{half}} \vee C_{2m\tau i} \notin \mathcal{T}_{\mathrm{exist}}$
$\mathrm{HALF}(i)$:	$\mathrm{HALF}(i) = 1$ if every agent joins only less than $t_{\max}/2$ interactions among $\Gamma_{2m\tau i}, \dots, \Gamma_{2m\tau(i+1)-1}$, otherwise $\mathrm{HALF}(i) = 0$.
$\#_{TI}(v, t_1, t_2)$:	the number of interactions among $\Gamma_{t_1}, \dots, \Gamma_{t_2}$ involving the token that agent v has in configuration C_{t_1} (The definition is given just after Lemma 6.)

Assumptions
$n \geq 3$
$t_{\max} \geq 8\delta \max(d,\ \lceil 2 \log mn^3 d \rceil)$
$t_{\mathrm{virus}} = t_{\max}/2$
$4\delta t_{\max} \lceil \log n \rceil \leq t_{\mathrm{epi}} \leq \tau e^{\tau}/(9n)$

Before proving equations (1) and (2), we define ten sets of configurations:

$$\begin{aligned}
\mathcal{L}_{\mathrm{one}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_L(C) = 1\},\\
\mathcal{T}_{\mathrm{one}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_T(C) = 1\},\\
\mathcal{L}_{\mathrm{exist}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_L(C) \geq 1\},\\
\mathcal{T}_{\mathrm{exist}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \#_T(C) \geq 1\},\\
\mathcal{L}_{\mathrm{half}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{timer_L} > t_{\max}/2\},\\
\mathcal{T}_{\mathrm{half}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{timer_T} > t_{\max}/2\},\\
\mathcal{V}_{\mathrm{same}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \exists u, \forall v \in V,\ C(u).\mathtt{leader} = \top\\
&\qquad\qquad\qquad\qquad \wedge (C(v).\mathtt{timer_V} > 0 \Rightarrow C(u).\mathtt{color} = C(v).\mathtt{color})\},\\
\mathcal{V}_{\mathrm{zero}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{timer_V} = 0\},\\
\mathcal{E}_{\mathrm{half}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{AR}}) \mid \forall v \in V,\ C(v).\mathtt{token} = \top \Rightarrow C(v).\mathtt{timer_E} > t_{\mathrm{epi}}/2\},\\
\mathcal{S}_{\mathrm{AR}} &= \mathcal{L}_{\mathrm{one}} \cap \mathcal{T}_{\mathrm{one}} \cap \mathcal{L}_{\mathrm{half}} \cap \mathcal{T}_{\mathrm{half}} \cap \mathcal{V}_{\mathrm{same}} \cap (\mathcal{E}_{\mathrm{half}} \cup \mathcal{V}_{\mathrm{zero}})
\end{aligned}$$

where $\#_L(C)$ and $\#_T(C)$ denote the number of leaders and tokens in configuration C, respectively. Note that $\mathcal{V}_{\mathrm{same}}$ is the set of configurations where there exists a leader agent such that every agent with a virus has the same color as the leader, and $\mathcal{E}_{\mathrm{half}}$ is the set of configurations where every token has the epidemic timer whose value is greater than $t_{\mathrm{epi}}/2$.

First, we analyze the expected holding time. Let $C_0 \in \mathcal{S}_{\mathrm{AR}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \dots$. To prove (2), it is sufficient to show that both (i) $C_0, \dots, C_{8m\delta\tau\lceil \log n \rceil} \in LE$ and (ii) $C_{8m\delta\tau\lceil \log n \rceil} \in \mathcal{S}_{\mathrm{AR}}$ hold with probability no less than $p_{\mathrm{suc}} = 1 - O(n\delta \log n \cdot e^{-\tau})$. Then, letting $A = \min_{C_0 \in \mathcal{S}_{\mathrm{AR}}} \mathrm{EHT}_{P_{\mathrm{AR}}}(C_0, LE)$, we have $A \geq 8m\delta\tau\lceil \log n \rceil p_{\mathrm{suc}}/(1-p_{\mathrm{suc}}) = \Omega(\tau e^{\tau})$, since $A \geq p_{\mathrm{suc}}(8m\delta\tau\lceil \log n \rceil + A)$. We give five conditions such that satisfying all the conditions leads to above conditions (i) and (ii) (Lemma 10). After that, we analyze the probability that all the five conditions hold and prove that the probability is no less than $1 - O(n\delta \log n \cdot e^{-\tau})$.

We define three predicates $\mathrm{PROP}_L(i)$, $\mathrm{PROP}_T(i)$ and $\mathrm{HALF}(i)$ for any $i \geq 0$: $\mathrm{PROP}_L(i) = 1$ if $C_{2m\tau(i+1)} \in \mathcal{L}_{\mathrm{half}}$ or $C_{2m\tau i} \notin \mathcal{L}_{\mathrm{exist}}$, otherwise $\mathrm{PROP}_L(i) = 0$; $\mathrm{PROP}_T(i) = 1$ if $C_{2m\tau(i+1)} \in \mathcal{T}_{\mathrm{half}}$ or $C_{2m\tau i} \notin \mathcal{T}_{\mathrm{exist}}$, otherwise $\mathrm{PROP}_T(i) = 0$; $\mathrm{HALF}(i) = 1$ if every agent joins less than $t_{\max}/2$ interactions among $\Gamma_{2m\tau i}, \dots, \Gamma_{2m\tau(i+1)-1}$, otherwise $\mathrm{HALF}(i) = 0$. Intuitively, $\mathrm{PROP}_L(i) = 1$ ($\mathrm{PROP}_T(i) = 1$) means that high value of $\mathtt{timer_L}$ ($\mathtt{timer_T}$) propagates from a leader (a token, respectively) to all the agents during $2m\tau$ interactions, and $\mathrm{HALF}(i) = 1$ means every agent does not interact so much during $2m\tau$ interactions. Note that $\mathrm{PROP}_L(i) = 1$ ($\mathrm{PROP}_T(i) = 1$) unconditionally holds when there exists no leader (token, respectively) in $C_{2m\tau i}$. In addition, we define binary random variable $TO_L(C_0, t_1, t_2)$ and $TO_T(C_0, t_1, t_2)$ for integers t_1 and t_2, ($0 \leq t_1 \leq t_2$) as follows: $TO_L(C_0, t_1, t_2) = 1$ if there exists integer i ($t_1 \leq i < t_2$) satisfying $\#_L(C_i) < \#_L(C_{i+1})$, otherwise $TO_L(C_0, t_1, t_2) = 0$; $TO_T(C_0, t_1, t_2) = 1$ if there exists integer i ($t_1 \leq i < t_2$) satisfying $\#_T(C_i) < \#_T(C_{i+1})$, otherwise $TO_T(C_0, t_1, t_2) = 0$. Intuitively, variable $TO_L(C_0, t_1, t_2)$ (variable $TO_T(C_0, t_1, t_2)$) represents whether an interaction among $\Gamma_{t_1}, \dots, \Gamma_{t_2-1}$ trigger the leader timeout (the token timeout, respectively) or not.

▶ **Lemma 4.** *Let $C_0 \in \mathcal{L}_{\mathrm{half}} \cap \mathcal{L}_{\mathrm{exist}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \dots$. We have $C_{2m\tau} \in \mathcal{L}_{\mathrm{half}}$ and $TO_L(C_0, 0, 2m\tau) = 0$ if $\mathrm{PROP}_L(0) = \mathrm{HALF}(0) = 1$.*

Proof. Since there exists a leader in C_0, $\mathrm{PROP}_L(0) = 1$ assures $C_{2m\tau} \in \mathcal{L}_{\mathrm{half}}$. Assumptions $C_0 \in \mathcal{L}_{\mathrm{half}}$ and $\mathrm{HALF}(0) = 1$ assures that the leader timeout does not happen by $\Gamma_0, \dots, \Gamma_{2m\tau-1}$. ◀

▶ **Corollary 5.** *Let $C_0 \in \mathcal{L}_{\mathrm{half}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \dots$. Let $k \geq 1$ be any integer. We have $C_{2m\tau k} \in \mathcal{L}_{\mathrm{half}}$ and $TO_L(C_0, 0, 2m\tau k) = 0$ if $\mathrm{PROP}_L(i) = \mathrm{HALF}(i) = 1$ and $C_{2m\tau i} \in \mathcal{L}_{\mathrm{exist}}$ hold for all $i = 0, 1, \dots, k-1$.*

Once a token exist in the population, the number of tokens never become zero after that. Hence, we have a simpler lemma as for the token timeout.

▶ **Lemma 6.** *Let $C_0 \in \mathcal{T}_{\mathrm{half}} \cap \mathcal{T}_{\mathrm{exist}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \dots$. Let $k \geq 1$ be any integer. We have $C_{2m\tau k} \in \mathcal{T}_{\mathrm{half}} \cap \mathcal{T}_{\mathrm{exist}}$ and $TO_T(C_0, 0, 2m\tau k) = 0$ if $\mathrm{PROP}_T(i) = \mathrm{HALF}(i) = 1$ holds for all $i = 0, 1, \dots, k-1$.*

For agent $v \in V$ and integers t_1 and t_2, ($0 \leq t_1 < t_2$), we define $\#_{TI}(v, t_1, t_2) = |\{t \in [t_1 + 1, t_2] \mid v_t \neq v_{t-1}\}|$ where $v_{t_1} = v$, and

$$v_t = \begin{cases} u & \text{if } \Gamma_{t-1} = (u, v_{t-1}) \\ w & \text{if } \Gamma_{t-1} = (v_{t-1}, w) \\ v_{t-1}. & \text{otherwise} \end{cases}$$

for $t > t_1$. Random variable $\#_{TI}(v, t_1, t_2)$ has a intuitive meaning if v has a token when interaction Γ_t occurs: Intuitively, $\#_{TI}(v, t_1, t_2)$ represents the number of interactions that the token involves during $\Gamma_{t_1}, \dots, \Gamma_{t_2-1}$ (or the number of times the token moves during the period).

▶ **Lemma 7.** *Let $C_0 \in \mathcal{S}_{\mathrm{AR}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \ldots$. Let v_T be the agent that has the unique token in configuration C_0, and $t \ge 0$ be a non-negative integer. Then, we have $C_i \in \mathcal{V}_{\mathrm{same}}$ for all $i = 0, 1, \ldots, t$ if we have $\#_{TI}(v_T, 0, t) < t_{\mathrm{epi}}/2$ and $C_i \in \mathcal{T}_{\mathrm{one}}$ for all $i = 0, 1, \ldots, t$.*

Proof. Let v_L be the unique leader in configuration C_0, and we assume that the color of v_L and all agents with viruses are black without loss of generality (Note that $C_0 \in \mathcal{V}_{\mathrm{same}}$). Since $C_0 \in \mathcal{E}_{\mathrm{half}} \cup \mathcal{V}_{\mathrm{zero}}$, we prove the lemma for two cases $C_0 \in \mathcal{E}_{\mathrm{half}}$ and $C_0 \in \mathcal{V}_{\mathrm{zero}}$. In case $C_0 \in \mathcal{E}_{\mathrm{half}}$, the epidemic timer of the unique token never becomes zero in $C_0, \ldots, C_t$ because $\#_{TI}(v_T, 0, t) < t_{\mathrm{epi}}/2$. Therefore, a new virus is not created during $C_0, \ldots, C_t$, which assures that v_l and all agents with viruses are still black in $C_0, \ldots, C_t$. Thus, we have $C_i \in \mathcal{V}_{\mathrm{same}}$ for all $i = 1, 2, \ldots, t$. In case $C_0 \in \mathcal{V}_{\mathrm{zero}}$, the virus creation happens at most once during $C_0, \ldots, C_t$ because $\#_{TI}(v_T, 0, t), < t_{\mathrm{epi}}/2$ and $C_i \in \mathcal{T}_{\mathrm{one}}$ for all $i = 0, 1, \ldots, t$. If the virus creation does not happen, $C_i \in \mathcal{V}_{\mathrm{zero}} \cap \mathcal{L}_{\mathrm{exist}} \subseteq \mathcal{V}_{\mathrm{same}}$ holds for all $i = 0, 1, \ldots, t$. If a leader meets a token with an epidemic timer of value zero and creates a new virus, the virus propagates from agent to agent. However, the virus makes all infected agents the same color as the leader that creates the virus, which assures $C_i \in \mathcal{V}_{\mathrm{same}}$ for all $i = 0, 1, \ldots, t$. ◀

The following lemma is directly obtained from Corollary 5 and Lemma 7.

▶ **Lemma 8.** *Let $C_0 \in \mathcal{S}_{\mathrm{AR}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \ldots$. Let v_T be the agent that has the unique token in configuration C_0, and $k \ge 0$ be any integer. Then, we have $C_{2m\tau k} \in \mathcal{L}_{\mathrm{half}} \cap \mathcal{V}_{\mathrm{same}}$ and $C_i \in \mathcal{L}_{\mathrm{one}}$ for all $i = 0, 1, \ldots, 2m\tau k$ if we have $\mathrm{PROP}_L(j) = \mathrm{HALF}(j) = 1$ for all $j = 0, 1, \ldots, k-1$, $\#_{TI}(v_T, 0, 2m\tau k) < t_{\mathrm{epi}}/2$, and $C_i \in \mathcal{T}_{\mathrm{one}}$ for all $i = 0, 1, \ldots, 2m\tau k$.*

We define the first round time $\mathrm{RT}_\Gamma(1)$ as the minimum t satisfying $\forall e \in E,\ 0 \le \exists t' \le t,\ \Gamma_{t'} = e$. For any $i \ge 2$, we define the i-th round time $\mathrm{RT}_\Gamma(i)$ as the minimum t satisfying $\forall e \in E,\ \mathrm{RT}_\Gamma(i-1) < \exists t' \le t,\ \Gamma_{t'} = e$.

▶ **Lemma 9.** *Let $C_0 \in \mathcal{S}_{\mathrm{AR}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \ldots$. Let $t \ge 0$ be any integer. We have $C_t \in \mathcal{E}_{\mathrm{half}} \cup \mathcal{V}_{\mathrm{zero}}$ if we have $\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) < t$, $\#_{TI}(v_T, 0, t) < t_{\mathrm{epi}}/2$, and $\#_T(C_i) = 1$ for all $i = 0, 1, \ldots, t$.*

Proof. If a new virus is not created among $\Gamma_0, \ldots, \Gamma_t$, then all viruses in the initial configuration vanish during the period since each round decreases the maximum value of $\mathtt{timer_V}$ by at least one. Thus, $C_t \in \mathcal{V}_{\mathrm{zero}}$ holds. If some agent v creates a new virus at $\Gamma_{t'}$, then the epidemic timer of the unique token are reset at the same time. (Note that the unique token always exist in the population by the assumption of the lemma.) Thus, we have $C_{t'}(v).\mathtt{timer_E} = t_{\mathrm{epi}}$. Since $\#_{TI}(v, t', t) \le \#_{TI}(v_T, 0, t) < t_{\mathrm{epi}}/2$, the epidemic timer of the unique token is no less than $t_{\mathrm{epi}} - t_{\mathrm{epi}}/2 = t_{\mathrm{epi}}/2$, which means $C_t \in \mathcal{E}_{\mathrm{half}}$. ◀

▶ **Lemma 10.** *Let $C_0 \in \mathcal{S}_{\mathrm{AR}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \ldots$. Let v_T be the agent that has the unique token in configuration C_0. Then, we have both $C_0, \ldots, C_{8m\delta\tau\lceil\log n\rceil} \in LE$ and $C_{8m\delta\tau\lceil\log n\rceil} \in \mathcal{S}_{\mathrm{AR}}$ if the following conditions hold:*

(A) $\#_{TI}(v_T, 0, 8m\delta\tau\lceil\log n\rceil) < t_{\mathrm{epi}}/2$,
(B) $\mathrm{PROP}_L(i) = 1$ *for all* $i = 0, 1, \ldots, 4\delta\lceil\log n\rceil - 1$,
(C) $\mathrm{PROP}_T(i) = 1$ *for all* $i = 0, 1, \ldots, 4\delta\lceil\log n\rceil - 1$,
(D) $\mathrm{HALF}(i) = 1$ *for all* $i = 0, 1, \ldots, 4\delta\lceil\log n\rceil - 1$, *and*
(E) $\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) < 8m\delta\tau\lceil\log n\rceil$.

Proof. Assigning $k = 4\delta\lceil\log n\rceil$, we obtain $C_{8m\delta\tau\lceil\log n\rceil} \in \mathcal{T}_{\mathrm{half}}$ and $C_j \in \mathcal{T}_{\mathrm{one}}$ for all $j = 0, 1, \ldots, 8m\delta\tau\lceil\log n\rceil$ by Lemma 6 and Conditions (C) ad (D). From Lemma 8 and Conditions

(A), (B), and (D), the unique token assures that $C_{8m\delta\tau\lceil\log n\rceil} \in \mathcal{L}_{\mathrm{half}} \cap \mathcal{V}_{\mathrm{same}}$ and $C_j \in \mathcal{L}_{\mathrm{one}}$ holds for $j = 0, 1, \dots, 8m\delta\tau\lceil\log n\rceil$. Note that $C_j \in \mathcal{L}_{\mathrm{one}}$ $(j = 0, 1, \dots, 8m\delta\tau\lceil\log n\rceil)$ guarantees not only that the number of leaders is one, but also that the unique leader is stable (i.e. $\exists v \in V, \forall i \in [0, 8m\delta\tau\lceil\log n\rceil], C_i(v).leader = \top$) because P_{AR} does not move the leader role from agent to agent at any one interaction. Hence, we have $C_0, \dots, C_{8m\delta\tau\lceil\log n\rceil} \in LE$. We have $C_{8m\delta\tau\lceil\log n\rceil} \in \mathcal{E}_{\mathrm{half}} \cup \mathcal{V}_{\mathrm{zero}}$ from Lemma 9, Condition (A), Condition (E), and $C_j \in \mathcal{T}_{\mathrm{one}}$ for all $j = 0, 1, \dots, 8m\delta\tau\lceil\log n\rceil$. Thus, we have shown that $C_{8m\delta\tau\lceil\log n\rceil} \in \mathcal{L}_{\mathrm{one}} \cap \mathcal{T}_{\mathrm{one}} \cap \mathcal{L}_{\mathrm{half}} \cap \mathcal{T}_{\mathrm{half}} \cap \mathcal{V}_{\mathrm{same}} \cap (\mathcal{E}_{\mathrm{half}} \cup \mathcal{V}_{\mathrm{zero}}) \subseteq \mathcal{S}_{\mathrm{AR}}$ ◀

▶ **Lemma 11.** *Let $C_0 \in \mathcal{T}_{\mathrm{one}}$ and $\Xi_{P_{\mathrm{AR}}}(C_0) = C_0, C_1, \dots$. Let v_T be the agent that has the unique token in configuration C_0. Then, we have* $\Pr(\#_{TI}(v_T, 0, 8m\delta\tau\lceil\log n\rceil) < t_{\mathrm{epi}}/2) \geq 1 - e^{-\delta\tau}$.

Proof. For every $i \geq 0$, the token joins interaction Γ_i with probability at most δ/m regardless of the location of the token in C_i because any agent has at most δ edges. Thus, $\#_{TI}(v_T, 0, 8m\delta\tau\lceil\log n\rceil)$ is bounded by binomial random variable $X \sim B(8m\delta\tau\lceil\log n\rceil, \delta/m)$. We have

$$\begin{aligned}
\Pr(X \geq t_{\mathrm{epi}}/2) &\leq \Pr(X \geq 16\delta^2\tau\lceil\log n\rceil) && \because t_{\mathrm{epi}} \geq 32\delta^2\tau\lceil\log n\rceil \\
&= \Pr(X \geq 2\mathbf{E}[X]) \\
&\leq e^{-\mathbf{E}[X]/3} && \text{(By Chernoff Bound of Lemma 2 with } \kappa = 1) \\
&= e^{-8\delta^2\tau\lceil\log n\rceil/3} \\
&\leq e^{-\delta\tau},
\end{aligned}$$

which gives the lemma. ◀

▶ **Lemma 12.** $\Pr(\mathrm{PROP}_L(i) = 1) \geq 1 - 2ne^{-\tau}$ *for any $i \geq 0$.*

Proof. We assume $i = 0$ without loss of generality, and prove $\Pr(\mathrm{PROP}_L(0) = 1) \geq 1 - 2ne^{-\tau}$. We have $\mathrm{PROP}_L(0) = 1$ by the definition of PROP_L if no leader exists in C_0. Thus, it suffices to show $\Pr(C_{2m\tau}(v).\mathtt{timer_L} > t_{\max}/2) \geq 1 - 2e^{-\tau}$ for any agent $v \in V$ in case $C_0 \in \mathcal{L}_{\mathrm{exist}}$. Let v_L be a leader agent in C_0. We denote the shortest path from v_L to v by $(v_0, v_1, \dots, v_s)$ where $v_0 = v_L$, $v_s = v$, $0 \leq s \leq d$ and $(v_{j-1}, v_j) \in E$ for all $j = 1, 2, \dots, s$. For any $t = 0, 1, \dots, 2m\tau$, we define $v_{\mathrm{head}}(t)$ as v_h with maximum $h \in [1, s]$ such that there exist $t_1, t_2, \dots, t_h$ satisfying $0 \leq t_1 < t_2 < \cdots < t_h < t$ and $\Gamma_{t_j} \in \{(v_{j-1}, v_j), (v_j, v_{j-1})\}$ for $j = 1, 2, \dots, h$. We define $v_{\mathrm{head}}(t) = v_0$ if such h does not exist. Intuitively, $v_{\mathrm{head}}(t)$ is the head of the agents in path $(v_0, v_1, \dots, v_l)$ to which a large value of $\mathtt{timer_L}$ is propagated from v_L to v. (Remind that v_L resets $\mathtt{timer_L}$ to $t_{\max}$.) We define $J(t)$ as the number of integers $j \in [0, \dots, 2m\tau - 1]$ such that $v_{\mathrm{head}}(j)$ joins interaction Γ_j. Intuitively, $J(t)$ is the number of interactions that the head agent joins among $\Gamma_0, \dots, \Gamma_{2m\tau-1}$. Obviously, we have $C_t(v_{\mathrm{head}}(t)).\mathtt{timer} \geq t_{\max} - J(t)$ for any $t = 0, 1, \dots, 2m\tau$.

In what follows, we prove $\Pr(v_{\mathrm{head}}(2m\tau) = v) \geq 1 - e^{-\tau}$ and $\Pr(J(2m\tau) < t_{\max}/2) \geq 1 - e^{-\tau}$, which give $\Pr(C_{2m\tau}(v).\mathtt{timer} > t_{\max}/2) \geq 1 - 2e^{-\tau}$. For any $j = 1, \dots, s$, a pair v_{j-1} and v_j interacts with probability $2/m$ at each interaction. Hence, we can say each interaction makes v_{head} forward with probability $2/m$. Therefore, by letting Z be a binomial random variable such that $Z \sim B(2m\tau, 2/m)$, we have

$$\begin{aligned}
\Pr(v_{\text{head}}(2m\tau) = v) &= 1 - \Pr(Z < s)\\
&\geq 1 - \Pr(Z < d)\\
&\geq 1 - \Pr\left(Z < \frac{1}{4} \cdot \mathbf{E}[Z]\right) && \because d \leq \tau = \frac{1}{4} \cdot \mathbf{E}[Z]\\
&\geq 1 - e^{-9\mathbf{E}[Z]/32} && \text{(By Chernoff bound of Lemma 3 with } \kappa = \frac{3}{4}\text{)}\\
&> 1 - e^{-\tau}.
\end{aligned}$$

The probability that $v_{\text{head}}(t)$ joins interaction Γ_t is at most δ/m regardless of t. Hence, by letting Z' be a binomial random variable such that $Z' \sim B(2m\tau, \delta/m)$, we have

$$\begin{aligned}
\Pr(J(2m\tau) < t_{\max}/2) &> 1 - \Pr(Z' \geq t_{\max}/2)\\
&> 1 - \Pr(Z' \geq 2\mathbf{E}[Z'])\\
&> 1 - e^{-\mathbf{E}[Z']/3} && \text{(By Chernoff bound of Lemma 2 with } \kappa = 1\text{)}\\
&= 1 - e^{-2\delta\tau/3}\\
&> 1 - e^{-\tau}. && \because \delta \geq 2
\end{aligned}$$

Thus, we have shown $\Pr(C_{2m\tau}(v).\texttt{timer}_{\texttt{L}} > t_{\max}/2) \geq 1 - 2e^{-\tau}$. ◀

▶ **Lemma 13.** $\Pr(\text{PROP}_T(i) = 1) \geq 1 - 2ne^{-\tau}$ *for any* $i \geq 0$.

Proof. The same argument as the proof of Lemma 12 gives the lemma. ◀

▶ **Lemma 14** (in [14]). *The probability that every* $v \in V$ *interacts only less than* $t_{\max}/2$ *times during* $2m\tau$ *interactions is at least* $1 - ne^{-\tau}$.

Proof. For any $v \in V$ and $i \geq 0$, v joins interaction Γ_i with probability at most δ/m. Thus, the number of interactions v joins during the $2m\tau$ interactions is bounded by binomial random variable $X \sim B(2m\tau, \delta/m)$. Applying Chernoff bound of Lemma 2 with $\kappa = 1$, we have

$$\begin{aligned}
\Pr(X \geq t_{\max}/2) &\leq \Pr(X \geq 2\mathbf{E}[X]) && \because t_{\max} \geq 8\delta\tau\\
&\leq e^{-\mathbf{E}[X]/3}\\
&= e^{-2\delta\tau/3} && \text{(By Chernoff Bound of Lemma 2 with } \kappa = 1\text{)}\\
&\leq e^{-\tau}. && \because \delta \geq 2
\end{aligned}$$

Summing up the probabilities for all $v \in V$ gives the lemma. ◀

▶ **Lemma 15** (in [14]). $\Pr(\text{HALF}(i) = 1) \geq 1 - ne^{-\tau}$ *for any* $i \geq 0$.

Proof. Each interaction is independent. Thus, Lemma 14 gives the lemma. ◀

▶ **Lemma 16** (in [14]). $\Pr(\text{RT}_\Gamma(i) < im(1 + \lceil \log n \rceil)) \geq 1 - ne^{-i/4}$ *holds for any* $i \geq 1$.

Proof. The proof in [14] can be used with slight modification. ◀

▶ **Lemma 17.** $\Pr(\text{RT}_\Gamma(t_{\text{virus}}) < 8m\delta\tau\lceil \log n \rceil) \geq 1 - ne^{-\delta(\tau+1)}$ *holds.*

Proof. By Lemma 16, we have

$$\begin{aligned}\Pr(\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) < 8m\delta\tau\lceil\log n\rceil) &\geq \Pr(\mathrm{RT}_\Gamma(4\delta(1+\tau)) < 8m\delta\tau\lceil\log n\rceil)\\ &\geq \Pr(\mathrm{RT}_\Gamma(4\delta(1+\tau)) < 4m\delta(1+\tau)(1+\lceil\log n\rceil))\\ &\geq 1 - ne^{-\delta(\tau+1)}\end{aligned}$$

where we use $t_{\mathrm{virus}} \leq 4\delta(1+\tau)$ for the first inequality, and use $(1+\tau)(1+\lceil\log n\rceil) \leq 2\tau\lceil\log n\rceil$ when $\tau \geq 3$ and $n \geq 3$ for the second inequality. (Note that $\tau \geq \lceil 2\log mn^3 d\rceil \geq 10$.) ◀

▶ **Lemma 18.** $\min_{C\in\mathcal{S}_{\mathrm{AR}}} \mathrm{EHT}_{P_{\mathrm{AR}}}(C, LE) = \Omega(\tau e^\tau)$.

Proof. Probability p_{suc}, discussed in the beginning of this section, is at least $1 - e^{-\delta\tau} - 4\delta\lceil\log n\rceil(2ne^{-\tau} + 2ne^{-\tau} + ne^{-\tau}) - ne^{-\delta(\tau+1)} \geq 1 - 22n\delta\lceil\log n\rceil e^{-\tau}$ by Lemmas 10, 11, 12, 13, 15, and 17, which leads to the lemma. ◀

Next, we analyze the expected convergence time.

▶ **Lemma 19.** $\max_{C\in\mathcal{C}_{\mathrm{all}}} \mathrm{ECT}_{P_{\mathrm{AR}}}(C, \mathcal{S}_{\mathrm{AR}}) = O(mt_{\mathrm{epi}} + mn^3 d)$.

Proof Sketch. In an execution of P_{AR} , the population converges to $\mathcal{S}_{\mathrm{AR}}$ starting from any configuration through the following convergence steps: (i) a token is created even when no token exists, (ii) the number of tokens become one, i.e. the unique token is elected, (iii) all viruses vanish from the population, (iv) the epidemic timer of the unique token becomes zero, (v) the unique token meets a leader and a new virus is created, (vi) a newly created virus propagates to the whole population and changes all agents to the ones with the same color (Let the color be black without loss of generality), (vii) the epidemic timer of the unique token becomes zero, (viii) the unique token meets a leader and a new virus is created, (ix) a newly created virus propagates to the whole population and makes all agents white, which kills all leaders other than the leader that creates the virus, and the population enters $\mathcal{S}_{\mathrm{AR}}$. Steps (ii), (iv) and (vii) require the dominant number of interactions. We will prove that the expected number of interactions until two tokens meet is $O(mn^2 d)$ in Lemma 20. The number of tokens is at most n, and the token timeout, which is the only event that increases the number of tokens, rarely happens once a token exists. Hence, the expected number of interactions Step (ii) requires is $O(mn^3 d)$. The expected number of interactions Step (iv) and (vii) require is $O(mt_{\mathrm{epi}})$ because the epidemic timer decreases by one as the token joins an interaction, and the unique token joins each interaction Γ_t with probability at least $2/m$. ◀

▶ **Lemma 20.** *Let C_0 be a configuration where two or more tokens exist. In execution $\Xi_{P_{\mathrm{AR}}}(C_0)$, the expected number of interactions until two tokens meet is at most $mn^2 d/2$.*

Proof. Let $u, v \in V$ be the distinct two agents both of which have tokens in C_0. We analyze the expected number of interactions until the two tokens meet. (One of the two tokens may vanish by meeting another token, however, this just reduces the expected number of interactions until any two tokens meet.) Consider the pair of random walks by the two tokens on population G, i.e. a Markov chain (u_t, v_t) in which the states of the chain are pairs of the agents in G. We denote $(a,b) \to (c,d)$ for agents $a,b,c,d \in V$ if $(a,c) \in E \wedge b = d$, or $(b,d) \in E \wedge a = c$, or $(a,b) \in E \wedge a = d \wedge b = c$. For any two states x and y, the transition probability $P_{x,y}$ of the chain is given by $P_{x,y} = 2/m$ if $x \to y$, $P_{x,y} = 1 - (2/m)|\{z \mid x \to z\}|$ if $x = y$, otherwise $P_{x,y} = 0$. The symmetry structure of the chain ($P_{x,y} = P_{y,x}$) gives $\sum_x P_{x,y} = 1$ for all state y. Thus, $\pi = (\pi(x_1), \pi(x_2), \dots, \pi(x_{n(n-1)})) = \{n(n-1)\}^{-1}(1,1,\dots,1)$ is the stationary distribution of the chain ($\pi P = \pi$) where $x_1, x_2, \dots, x_{n(n-1)}$ are all the states of the chain (i.e. all pairs of token locations). We denote the expected number of transition steps from state x to state y by $h_{x,y}$. We have $h_{y,y} = 1/\pi(y) = n(n-1)$ for any state y. We also have $h_{y,y} = 1 + \sum_{y \to z}(2/m) \cdot h_{z,y}$. Hence, we obtain $\sum_{y \to z} h_{z,y} = n(n-1)m/2 - m/2$. Thus, we have $h_{x,y} \leq mn^2/2$ for any states x and y satisfying $x \to y$. Let $w_0, w_1, \dots, w_l$ ($w_0 = u, w_l = v, l \leq d$) be the shortest path from u to v. The expected time until the two token meet is bounded by $\left(\sum_{i=0}^{l-2} h_{(w_i,w_l),(w_{i+1},w_l)}\right) h_{(w_{l-1},w_l),(w_l,w_{l-1})} \leq mn^2 d/2$. ◀

Lemmas 18 and 19 gives the following theorem.

▶ **Theorem 21.** *Protocol P_{AR} is an $(O(mt_{\mathrm{epi}} + mn^3 d), \Omega(\tau e^{\tau}))$- loosely-stabilizing leader election protocol for arbitrary graphs G when $t_{\max} \geq 8\delta \max(d, \lceil 2 \log mn^3 d \rceil)$, $t_{\mathrm{virus}} = t_{\max}/2$, and $4\delta t_{\max} \lceil \log n \rceil \leq t_{\mathrm{epi}} \leq \tau e^{\tau}/(9n)$.*

Therefore, given an upper bounds N of n and upper bound Δ of δ, we have a $(O(mn^3 d + mN\Delta^2 \log N), \Omega(Ne^N))$ loosely-stabilizing leader election protocol for arbitrary graphs if we assign $t_{\max} = 8\Delta \max(N, \lceil 12 \log N \rceil)$, $t_{\mathrm{virus}} = t_{\max}/2$, $t_{\mathrm{epi}} = 4\Delta t_{\max} \lceil \log N \rceil$.

5 Conclusion

We have presented a loosely-stabilizing leader election protocol for arbitrary undirected graphs in the population protocol model. It does not use agent identifiers nor random numbers unlike our previous protocols. Given upper bounds N of n and Δ of δ, the population reaches a safe configuration within $O(mn^3 d + mN\Delta^2 \log N)$ expected interactions, and after that, keeps a unique leader for $\Omega(Ne^N)$ expected interactions. The restriction to *undirected* graph is only for simplicity of complexity analysis, and P_{AR} works on arbitrary *directed* graphs without modifications.

References

1 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. doi:10.1007/s00446-005-0138-3.

2 D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. In *DISC*, pages 61–75, 2006.

3 D. Angluin, J. Aspnes, M. J Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):13, 2008.

4 J. Beauquier, P. Blanchard, and J. Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *OPODIS*, pages 38–52, 2013.

5 J. Beauquier, J. Burman, L. Rosaz, and B. Rozoy. Non-deterministic population protocols. In *OPODIS*, pages 61–75, 2012.

6 S. Cai, T. Izumi, and K. Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.

7 D. Canepa and M. G. Potop-Butucaru. Stabilizing leader election in population protocols, 2007. URL: http://hal.inria.fr/inria-00166632.

8 M. J. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *OPODIS*, pages 395–409, 2006. doi:10.1007/11945529_28.

9 T. Izumi. On space and time complexity of loosely-stabilizing leader election. In *SIROCCO*, 2015.

10 O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Mediated population protocols. *Theoretical Computer Science*, 412(22):2434–2450, 2011.

11 M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

12 R. Mizoguchi, H. Ono, S. Kijima, and M. Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25(6):451–460, 2012.

13 Y. Sudo, J. Nakamura, Y. Yamauchi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theoretical Computer Science*, 444:100–112, 2012.

14 Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-stabilizing leader election on arbitrary graphs in population protocols. In *OPODIS*, pages 339–354. Springer, 2014.

15 X. Xu, Y. Yamauchi, S. Kijima, and M. Yamashita. Space complexity of self-stabilizing leader election in population protocol based on k-interaction. In *SSS*, pages 86–97, 2013.

A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms*

Orr Tamir[1], **Adam Morrison**[2], **and Noam Rinetzky**[3]

1 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
`ortamir@post.tau.ac.il`

2 **Computer Science Department, Technion – Israel Institute of Technology, Haifa, Israel**
`mad@cs.technion.ac.il`

3 **Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel**
`maon@cs.tau.ac.il`

Abstract

Existing concurrent priority queues do not allow to update the priority of an element after its insertion. As a result, algorithms that need this functionality, such as Dijkstra's single source shortest path algorithm, resort to cumbersome and inefficient workarounds. We report on a heap-based concurrent priority queue which allows to change the priority of an element after its insertion. We show that the enriched interface allows to express Dijkstra's algorithm in a more natural way, and that its implementation, using our concurrent priority queue, outperform existing algorithms.

1998 ACM Subject Classification D.1.3 [Programming Techniques]:Concurrent Programming

Keywords and phrases priority queues, concurrent data structures, `changeKey`(), Dijkstra's single-source shortest path algorithm

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.15

1 Introduction

A priority queue data structure maintains a collection (multiset) of items which are ordered according to a *priority* associated with each item. Priority queues are amongst the most useful data structures in practice, and can be found in a variety of applications ranging from graph algorithms [21, 5] to discrete event simulation [8] and modern SAT solvers [4]. The importance of priority queues has motivated many concurrent implementations [1, 2, 11, 15, 16, 17, 23, 24, 25, 26]. These works all focus on the performance of two basic priority queue operations, which consequently are the only operations provided by concurrent priority queues: `insert`(d, p), which adds a data item d with priority p, and `extractMin`(), which removes and returns the highest-priority data item.[1]

It turns out, however, that important applications of priority queues, such as Dijkstra's single-source shortest path (SSSP) algorithm [5, 3], need to update the priority of an item after its insertion, i.e., *mutable priorities*. Parallelizing these algorithms requires working around the lack of mutable priorities in today's concurrent priority queues by inserting new

* This work was funded by EU FP7 project ADVENT (308830), ERC grant agreement no. [321174-VSSC], by Broadcom Foundation and Tel Aviv University Authentication Initiative, by Israel Science Foundation (grants 652/11, 1227/10 and 1749/14), and by Yad HaNadiv foundation.

[1] In this paper, we consider lower p values to mean higher priorities.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 15; pp. 15:1–15:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

items instead of updating existing ones, and then identifying and ignoring extracted items with outdated priorities [1] – all of which impose overhead. Sequential heap-based priority queues support mutable priorities [3], but concurrent heaps have been abandoned in favor of skiplist-based designs [17] whose `extractMin`() and `insert`() are more efficient and scalable. Thus, the pursuit of performance for the basic priority queue operations can, ironically, end up leading to worse *overall* performance for the parallel *client application*.

The principle driving this work is that we should design concurrent data structures with the *overall performance* of the *client* as the goal, even if this entails compromising on the performance of the individual data structure operations. We apply this principle to priority queues by implementing CHAMP, a Concurrent Heap with Mutable Priorities, which provides a `changeKey`() operation to update priorities of existing items. We use CHAMP to implement a parallel version of Dijkstra's SSSP algorithm, and our experimental evaluation shows that, as parallelism increases, CHAMP's efficient `changeKey`() operation improves the client overall performance by saving it from doing *wasted work* – the overhead that arises when working around the lack of `changeKey`() support in prior designs. This occurs despite the fact that CHAMP's `extractMin`() and `insert`() operations do not scale as well as in prior designs.

Contributions. To summarize, we make the following technical contributions:

1. We present CHAMP, a linearizable lock-based concurrent priority queue that supports mutable priorities. CHAMP is an adaptation of the concurrent heap-based priority queue of [11] to support the `changeKey`() operation.
2. We convert an existing parallel SSSP algorithm to utilize the `changeKey`() operation.
3. We implement and evaluate our algorithms.

Arguably, the more important contribution of this paper is the conceptual one: A call to pay more attention in the design of data structures and interfaces to the overall performance and programmatic needs of the client applications than to the standalone scalability of the supported operations, as at the end, the client is always right.

2 Priority Queues with Mutable Priorities

A *priority queue with mutable priorities* (PQMP) is a data structure for maintaining a multiset $\boldsymbol{A}$ of *elements*, where each element is a pair comprised of a *data item* d and a value k called *priority*.[2] A PQMP supports the following operations:

- `extractMin`(): Removes and returns the element which has the highest priority in the queue.[1] In case multiple elements have the highest priority, one of them is chosen arbitrarily. If the queue is *empty*, a special value is returned.
- `peek`(): Acts similarly to `extractMin`(), except that the chosen element is not removed.
- `insert`(d, k): Inserts into the queue an element comprised of a given data item d and *priority* k, and returns a unique *tag* e identifying the element. If the queue has reached its *full capacity*, the element is not inserted, and a special value is returned.
- `changeKey`(e, k): Sets the priority of element e to k. If e is not in the queue, the operation has no effect. (The latter behavior was chosen for the sake of simplicity. An alternative, could have been, e.g., to return a special value or to raise an exception.)

The use of *tags* to identify elements in the queue, instead of their data items, as done, e.g., in [3, Ch. 6.5], allows to store in queue multiple elements with the same data item.

[2] The term *key* is sometimes used instead of *priority*.

```
Element[0..Length] A
int Last

Lock[0..Length] L
class Element
  Priority key
  Data data
  int pos

  bool up
```

Figure 1 The data representation of the heap.

```
swap(i,j)
 temp = A[i]
 A[i] = A[j]
 A[j] = temp
 A[i].pos = i
 A[j].pos = j
```

```
leftChild(i)
 return 2*i
```

```
rightChild(i)
 return 2*i+1
```

```
parent(i)
 return ⌊ i/2 ⌋
```

Figure 2 Auxiliary procedures.

3 A Sequential Heap with Mutable Priorities

PQMPs can be implemented with a *binary heap* data structure. A binary heap is an almost complete binary tree that satisfies the *heap property*: for any node, the *key* of the node is less than or equal to the keys of its children, if they exist [3]. Binary heaps are often represented as arrays: the root is located at position 1, and the left and right children of the node at location i are located at positions $2i$ and $2i+1$, respectively. Position 0 is not used. Heaps support `extractMin`(), `peek`(), `insert`(), and `changeKey`() operations that map naturally to respective priority queue operations, if we use elements' priorities as keys.

In the following, we describe a sequential implementation of an array-based heap. The sequential implementation is fairly standard. Thus, our description mainly focuses on certain design choices that we made in the concurrent algorithm which can be explained in the simpler sequential settings.

Fig. 1 defines the type `Element`, and shows the data representation of a heap using two global variables: An array `A` and an integer `Last`. (Array `L` and the `up` field in elements are used only by the concurrent algorithm.) A heap with maximal capacity `Length` is comprised of an array `A` of pointers to `Elements` with `Length+1` entries and a counter `Last` which records the number of elements in the heap. We say that an element is *in* the heap if some entry in `A` points to it. We refer to the element pointed to by `A[1]` as the *root element*.

An element is comprised of three fields: `key` keeps the element's priority, `data` stores an application-specific data item, and `pos` records the *position* of the element in the heap: Given an element e, the position of an element e is the index of an entry in `A` which points to e, or -1 if e is not in the heap, i.e., if e.`pos` $\neq -1$ then `A[`e.`pos]` $= e$.

Figure 3 shows the pseudocode of a sequential heap. The operations use the auxiliary functions defined in Fig. 2. Each heap operation consists of two parts. First, it inspects, adds, removes, or changes an element. Then, because this change may violate the heap property, it *heapifies* the heap in order to restore the heap property. In the following, we describe how heap operations are implemented and then how heapifying is done. We use the $\cdot_{\mathsf{seq}}$ subscript to distinguish between the sequential operations and the concurrent ones.

- `peek`$_{\mathsf{seq}}$(): Returns the root element or *null* if the heap is empty.
- `insert`$_{\mathsf{seq}}(d, k)$: Returns *null* if the heap is *full*. Otherwise, it allocates and inserts a new element into the heap.[3] The element is placed at the `Last` $+ 1$ entry of `A`, which is at the lowest level of the heap, right after the last occupied position in the array. After the

[3] In this paper, we sidestep the difficult problem of concurrent safe memory reclamation [19, 18, 9], and assume that memory is recycled either by the client or by an underlying garbage collector [12].

```
peek_seq()
 return A[1]
```

```
extractMin_seq()
 min = A[1]
 ls = Last
 if (ls = 0)
  return null
 min.pos = -1
 if (ls = 1)
  A[1] = null
 else
  A[1] = A[ls]
  A[1].pos = 1
  A[ls] = null
  if (ls = 2)
    Last = 1
  else
    Last = ls - 1
    bubbleDown_seq(A[1])
 return min
```

```
insert_seq(key, data)
 if (Last = Length)
   return null
 e = new Element(
  key, data, Last+1)
 if(Last = 0)
   A[1] = e
   Last = 1
   unlock(L[1])
 else
   lock(L[Last + 1])
   e.up = true
   A[Last + 1] = elm
   Last = Last + 1
   unlock(L[1])
   bubbleUp(e)
 return e
```

```
changeKey_seq(e, k)
 if (e.key∉{1..Last})
   return false
 if (k < e.key)
   e.key = k
   bubbleDown_seq(e)
 else if (k > e.key)
   e.key = k
   bubbleUp_seq(e)
 return true
```

```
bubbleDown_seq(e)
 min = e.pos
 do
   i = min
   l = leftChild(i)
   r = rightChild(i)
   if (l ≤ Last)
     if (A[l].key < A[i].key)
       min = l
     if (A[r] ≠ null and
         A[r].key < A[min].key)
       min = r
     if (i ≠ min)
       swap(i, min)
 while(i ≠ min)
```

```
bubbleUp_seq(e)
 i = e.pos
 do
   par = parent(i)
   if(A[i].key < A[par].key)
     swap(i, par)
     i = par
 while (i  =  par)
```

Figure 3 Pseudo code of a *sequential* heap with mutable priorities. **Length-1** is the capacity of the heap. We assume that **changeKey$_{\text{seq}}$(e, k)** is invoked with **e≠null**.

operation completes its second phase (heapify), it returns a pointer to the new element as its tag.

- extractMin$_{\text{seq}}$(): Returns *null* if the heap is *empty*. Otherwise, it replaces the root element with the rightmost element in the tree, which is the last occupied position in the array. After the second part (heapify), the operation returns the previous root element.
- changeKey$_{\text{seq}}$(): Changes the key of the specified element e to k, if e is in the heap. Note that position field of an element is used to locate the entry in A pointing to it.

The second part of the operation restores the heap property by heapifying: In extractMin$_{\text{seq}}$(), we use bubbleDown$_{\text{seq}}$(), which shifts the root element whose key might become larger than its children down in the heap until the heap property is restored. In insert$_{\text{seq}}$(), we use bubbleUp$_{\text{seq}}$(), which carries an element up in the heap until its key is larger than that of its parent. Finally, changeKey$_{\text{seq}}$() uses bubbleDown$_{\text{seq}}$() or bubbleUp$_{\text{seq}}$() as appropriate. Note that when an element is being swapped, its position field is updated too and that when an element is removed from the heap, its position is set to -1.

4 Champ: A Concurrent Heap with Mutable Priorities

In this section, we present a concurrent PQMP data structure based on CHAMP, a concurrent heap with mutable priorities. At its core, CHAMP is an array-based binary heap, very much like the sequential algorithm described in the previous section. Synchronization is achieved using a fine-grained locking protocol, derived from the one used in [11] (see Sec. 4.3.)

CHAMP is implemented using the global variables shown in Fig. 1. Variables A and Last play the same role as in the sequential setting (see Sec. 3.) Variable L is an array of locks which contains one lock for every entry in A. Intuitively, lock L[i] is used to synchronize

```
extractMin()
 Lock(A[1])
 min = A[1]
 ls = Last
 if (ls == 0)
   unlock(L[1])
   return null
 A[1].pos = -1
 if (ls == 1)
   Last = 0
   A[1] = null
   unlock(L[1])
 else
   lock(L[ls])
   A[1] = A[ls]
   A[1].pos = 1
   A[ls] = null
   Last = ls - 1
   unlock(L[ls])
   if (ls == 2)
     unlock(L[1])
   else
     bubbleDown(A[1])
 return min
```

```
insert(key, data)
 lock(L[1])
 if (Last == Length)
   unlock(L[1])
   return null
 e = new Element(
    key, data, Last+1)
 if(Last == 0)
   e.up = false
   A[1] = e
   Last = 1
   unlock(L[1])
 else
   lock(L[Last + 1])
   e.up = true
   A[Last + 1] = e
   Last = Last + 1
   unlock(L[1])
   bubbleUp(e)
 return e
```

```
changeKey(e, k)
 while (lockElement(e))
   if (e.up)
     unlock(L[e.pos])
   else
     if (k < e.key)
       e.up = true
       e.key = k
       bubbleUp(e)
     else if (k > e.key)
       e.key = k
       bubbleDown(e)
     else
       unlock(L[e.pos])
     return
```

```
peek()
 lock(L[1])
 ret = A[1]
 unlock(L[1])
 return ret
```

Figure 4 The pseudo code of CHAMP, a *concurrent* priority queue with updatable key based on a binary heap. The concurrent heapifying procedures are presented in Fig. 5. Auxiliary procedures, e.g., `swap()`, are defined in Fig. 2.

accesses to the i-th entry of `A` and to the element `A[`i`]` points to. Lock `L[1]`, which we refer to as the *root lock*, is also used to protect the `Last` variable. A thread is allowed to modify a shared memory location only under the protection of the appropriate lock. Read accesses to the entries of array `A` and to variable `Last` should also be protected by a lock. In contrast, fields of elements can be read without a lock.[3]

Figures 4 and 5 show the pseudocode of our concurrent heap. CHAMP implements the interface of a PQMP. As expected, the concurrent operations provide the same functionality as the corresponding sequential counterparts, and, like them, also consist of two stages: First, every operation grabs the locks it requires and inspects, adds, removes, or changes the shared state. Then, it invokes `bubbleUp`(e) or `bubbleDown`(e) to *locally* restore the heap property.

The more interesting aspects of the first part of the operations are summarized below:

- `peek`(): Although `peek`() only reads a single memory location, it starts by taking the root lock. This is required because another thread might perform an `insert`() operation concurrently, which could lead to a state where the key of the root is *not* lower than that of its children. Returning such a root element would violate linearizability (see Sec. 4.1).
- `insert`(), `extractMin`(), and `changeKey`(): The first part of these operations is the same as that of their sequential counterparts, but with two exceptions:

  Element locking. The operations begin by taking locks. `changeKey`(e, k) takes the lock of the array entry pointing to e. (We explain the reason for using a loop later on.) All other operations grab the root lock. Also, the operations avoid calling the heapifying procedures in cases where the *global* heap property is guaranteed to hold after the change, e.g., when an element is inserted into an empty heap or when the last element in the heap is extracted.

  Signaling upward propagation. `insert`(e) and `changeKey`(e) set the `up` flag of e before invoking `bubbleUp`(e). This indicates that e is being propagated up the heap, which help synchronize concurrent `bubbleUp`() operations, as we shortly explain.

```
bubbleDown(\elm)
 min = e.pos
 do
  i = min
  l = LeftChild(i)
  r = RightChild(i)
  if (l ≤ Last)
    lock(L[l])
    lock(L[r])
    if (A[l] ≠ null)
      if (A[l].key < A[i].key)
        min = l
      if (A[r] != null and
          A[r].key < A[min].key)
        min = r
     if (i ≠ min)
       if(i == l)
         unlock(L[r])
       else
         unlock(L[l])
       swap(i, min)
       unlock(L(i))
 while(i ≠ min)
 unlock(L[i])
```

```
lockElement(\elm)
 while(true)
   i = e.pos
   if (i == -1)
     return false
   if (trylock(L[i]))
     if (i == e.pos)
       return true
     unlock(L[i])
```

```
bubbleUp(\elm)
 i = e.pos
 iLocked = true
 parLocked = false
 while (1 < i)
   par = Parent(i)
   parLocked = tryLock(L[par])
   if (parLocked)
     if (!A[par].up)
       if(A[i].key < A[par].key)
         swap(i, par)
       else
         A[i].up = false
         unlock(L[i])
         unlock(L[par])
         return
     else
       unlock(L[par])
       parLocked = false
   unlock(L[i])
   iLocked = false
   if (parLocked)
     i = par
     iLocked = true
   else
     iLocked = lockElement(e)
     i = e.pos
 e.up = false
 if (iLocked)
   unlock(L[e.pos])
```

Figure 5 Concurrent heapifying procedures.

The second part of every operation *locally* restores the heap property using bubbleUp() and bubbleDown(). The two shift elements up, respectively, down the heap until the heap property is *locally* restored: bubbleUp(e) stops when the key of e is bigger than that of its parent. bubbleDown(e) stops when the key of e is smaller than the keys of its children. Both operations stop if they detect that e was extracted from the heap.

Both bubbleDown() and bubbleUp() employ the hand-over-hand locking protocol [13] (also known as the *tree-locking* protocols), but they acquire the locks in different orders: bubbleDown() takes the lock of the children of a node e only after it holds the lock of e while bubbleUp() takes the lock of the parent of e only when it has e's lock. The hand-over-hand protocol ensures deadlock freedom when all the operations take their locks in the same order. However, if different orders are used, deadlock might occur. To prevent deadlocks, bubbleDown() takes locks using **tryLock**() instead of **lock**(), and in case the needed lock is not available it releases all its locks and then tries to grab them again.

An interesting case may happen when bubbleUp(e) attempts to get a hold of e's lock after its release: It is possible that the up-going element e have been pulled upwards by a concurrent down-going bubbleDown() operation. In fact, the element might have been removed from the heap all together. The auxiliary procedure lockElement(e) is thus used to locate a possibly relocated element. It repeatedly finds e's position in A using its position field and tries to lock the corresponding entry. lockElement(e) loops until it either obtained the lock protecting e's position, or until it finds that e has been extracted from the heap, indicated by having value -1 in its position field.

There is a tricky synchronization scenario that might occur when a bubbleUp(e) operation t_1 manages to lock an entry i pointing to an up-going element e'. (This might occur if the operation t_2 bubbling-up e' has not managed to bring e' to a position where its value is bigger than that of its parent, but had to release its locks to prevent a possible deadlock.) If e's key is bigger than that of e', t_1 might come to the wrong conclusion that it has managed to restore the heap-property and terminate. However, the property was restored with respect to an element, e, which is *not* in its "right" place. To ensure that such a scenario does not happen, `bubbleUp`(e) releases its lock when it detects that the parent of the element it is propagating is also being bubbled-up, indicated by its `up` flag.

▶ Note. Our algorithm supports concurrent priority changes of a given element e. These operations synchronize using the loop at the entry to `changeKey`(): A thread can enter the loop only if it manages to lock the position of element e. In case it detects an ongoing `bubbleUp`(e) operation, it releases the lock and retries. (Note that the lock cannot be obtained if there is an ongoing `bubbleDown`(e) operation). This ensures that there is only one thread that changes the key of an element. We note that in certain clients, e.g., Dijkstra's SSSP algorithm, the client threads prevent concurrent priority changes of a given element. In this cases, e.`up` is always false when we enter the loop.

4.1 Linearizability

Champ is a linearizable [10] priority queue with mutable keys. Intuitively, linearizability means that every operation seems to take effect instantaneously at some point between its invocation and response. In our case, these *linearization points* are as follows:

1. `peek`(), `extractMin`(), and `insert`() when invoked on a heap containing two or less elements: The point in time when the operation obtained the root lock.
2. `insert`() and `changeKey`(e) which decreased the priority of e: The linearization point happens during the call to `bubbleUp`(e). It is placed at the last configuration in which an entry in `A` pointed to e before its `up` field was set to false.
3. `changeKey`(e) which did not not find e in the heap, increased its priority or did not change it: The point in time in which the call to `lockElement`(e) returned.

Technically, the proof of linearization rests upon the following invariants:

(i) No element is stored in position 0.
(ii) The entries `A[1]`..`A[Last]` contain non-*null* values.
(iii) The value of every entry `A[Last+1]`$\cdots$`A[Length]` is *null*, except perhaps during the first part of `insert`() when `A[Last` $+$ `1]` might have the same non-null value as `A[1]`.
(iv) The position field `pos` of an element agrees with its position in the heap, i.e., if e.`pos` $= i \wedge 0 < i$ then `A[`e`].pos` $= i$, except perhaps during a `swap`() involving `A[`i`]`.
(v) If the i-th entry in the heap and its parent $j = \lfloor i/2 \rfloor$ are not locked and, in addition, `A[`i`].up` $=$ *false* and `A[`j`].up` $=$ *false* then `A[`j`].key` $\leq$ `A[`i`]`.

Most of the invariants are quite simple and rather easy to verify, in particular, when we recall that the global variables and the elements can be modified only when the thread holds the lock which protects both the entry and the element it points to. Note that if an element is pointed to by two entries then the same operation holds the two locks protecting these entries. The only time a thread may modify a field of an object without holding the respective lock is when it sets off the `up` field of an element which was removed from the heap. The key invariant is (v). It provides a local analogue of the heap property. Intuitively, it says that if an element violates the heap property then there exists an ongoing operation which is "responsible" for rectifying the violation. Furthermore, any apparent inconsistency that might occur due to the violation can be mitigated by the placement of the linearization point of the responsible operation in the global linearization order. For example, we can justify an `extractMin`() operation which returns an element e although the heap contains a non-root element e' which has a lower key than e by placing its linearization point before that of the operation responsible for inserting e' or reducing its key. Invariant (v) ensures that such an operation is still active when `extractMin`() takes possession of the root lock.

4.2 DeadLock-Freedom

Champ is deadlock-free. All the operations except `bubbleUp`() capture their locks according to a predetermined order, thus preventing deadlock by construction. `bubbleUp`() uses **tryLock**(), and releases its locks if the latter fails, thus avoiding deadlocks all together.

4.3 Comparison with Hunt's Algorithm [11]

Our priority queue is based on concurrent heap of Hunt et al. [11], as both use fine-grained locking to synchronize bottom-up `insert`()s and top-down `extractMin`()s. The main difference is the addition of the `changeKey`() operation. There are also some subtle differences in certain key aspects of the implementation.

- We use a different technique to prevent deadlocks between concurrent up-going `bubbleUp`()s and down-going `bubbleDown`()s: In [11], `insert`()s and `extractMin`()s takes locks in the same order. Specifically, they lock the parent before its child. Deadlock is prevented by having `insert`()s release their locks before they climb up the heap. In our algorithm, `insert`() and `changeKey`() take their locks in reverse order, thus possibly saving some redundant **unlock**() and re**lock**() operations. Deadlock is prevented using **tryLock**()s operations as explained in Sec. 4.2.
- In both techniques, an element e bubbled up the heap might change its position due to a down-going operation. In [11], the up-going operation propagating e finds it by climbing up the heap. In our case, we embed a position index inside the node which allows to locate it in a more direct fashion. The position index is particularly beneficial for `changeKey`(e) as it allows to efficiently check whether e is in the heap.
- Hunt reduces contention between `insert`() operations using a bit-reversal scheme to determine the index into which a new element is added. We use the standard scheme for insertions which maintains all the elements in a single contiguous part. We note that we can easily import their method into our algorithm.
- Finally, Hunt's priority queue is *not* linearizable, while ours is. The culprit is the `extractMin`() procedure which first removes the `Last` element from the heap and only then places it at the root. This allows for certain non-linearizable behaviors to occur. It is important to note, however, that there is no claim of linearizability in [11], and once the reason for the non-linearizable behavior is known, changing the algorithm to be linearizable is rather easy.

5 Case Study: Parallelizing Dijkstra's SSSP Algorithm

Important applications of priority queues, such as Dijkstra's single-source shortest path (SSSP) algorithm [5, 3] and Prim's minimal spanning tree (MST) algorithm [21, 3] need to update the priority of an item after its insertion. i.e., *mutable priorities*. In this work, we close the interface gap between sequential and concurrent priority queues by importing the `changeKey`() operation from the sequential setting to the concurrent one. To evaluate the benefits clients may gain by using the extended interface, we adapted a parallel version of Dijkstra's SSSP algorithm to use `changeKey`().

The SSSP problem is to find, given a (possibly weighted) directed graph and a designated source node s, the weight of the shortest path from s to every other node in the graph. For every node v, we refer to the weight of a shortest $s \rightsquigarrow u$ path as v's *distance* from s. The asymptotically fastest known sequential SSSP algorithm for arbitrary directed graphs with unbounded non-negative weights is Dijsktra's algorithm [5, 7].

Dijkstra's algorithm partitions the graph into *explored* nodes, whose distance from s is known, and *unexplored* nodes, whose distance may be unknown. Each node v is associated with its distance, $dist(v)$, which is represented as a field in the node. The algorithm computes the distances by iteratively exploring the edges in the frontier between explored and unexplored nodes. The initial distances are $dist(s) = 0$ and $dist(v) = \infty$ for every $v \neq s$. In each iteration, the algorithm picks the unexplored node v with the smallest associated distance, marks it as explored, and then *relaxes* every edge (v, u) by checking whether $d = dist(v) + w(v, u) < dist(u)$, and if so, updating $dist(u)$ to d. Notice that once $dist(v) \neq \infty$, it always holds the length of some path from s to u, and hence $dist(v)$ is an upper bound on the weight of the shortest $s \rightsquigarrow v$ path.

Dijkstra's algorithm can be implemented efficiently using a priority queue with a `changeKey`() operation [7]. The idea is to maintain a queue of offers, where an *offer* $\langle v, d\rangle$ indicates that there is an $s \rightsquigarrow v$ path of weight d. An offer $\langle v, d\rangle$ is enqueued by inserting an element into the queue with key d and data v. In every iteration, the algorithm extracts a minimal offer $\langle v, d\rangle$ from the queue using `extractMin`(), and for each edge (v, u) it either `insert`()s a new offer (if $dist(u) = \infty$) or uses `changeKey`() to decrease the key of the existing offer $\langle u, d'\rangle$ if $dist(v) + w(v, u) < dist(u) = d'$.

Using `changeKey()` to parallelize Dijkstra's algorithm. Dijkstra's algorithm can be parallelized by using a concurrent priority queue from which multiple threads dequeue offers and process them in parallel. However, the existing parallel algorithm must work around the lack of `changeKey`() support in prior concurrent priority queues, with adverse performance consequences. Sec. 5.1 details this problem and describes the way existing parallel SSSP algorithms work. Sec. 5.2 describes the way our adaptation of the parallel algorithm addresses this problem by using `changeKey`().

Concurrent *dist* updates. Both parallel algorithms described next must guarantee that when relaxing an edge (v, u), reading $dist(v)$ and the subsequent decreasing of $dist(v)$ happen atomically. Otherwise, an update to d might get interleaved between another thread's read of $dist(v)$ and subsequent update to $d' > d$, and thus be lost. This atomicity is typically realized by performing the update with a `compare-and-swap` operation [1, 14]. Our implementations, however, use per-node locking: if a thread decides to update $dist(v)$, it acquires v's lock, verifies that $dist(v)$ should still be updated, and then performs the update. This approach allows us to atomically update an additional $P(v)$ field, which holds the predecessor node on the shortest path to v [7], and thus computes the shortest paths in addition to the distances. We omit the details of this, which are standard.

5.1 ParDijk: A Parallel version of Dijkstra's SSSP Algorithm based on a Concurrent Priority Queue

A natural idea for parallelizing Dijkstra's algorithm is to use a concurrent priority queue and thereby allow multiple threads to dequeue and process offers in parallel. Because existing concurrent priority queues do not support `changeKey`(), doing this requires adapting the algorithm to use inserts instead of `changeKey`() when relaxing edges [1, 14].

```
Graph (E,V,w)
done[1..TNum] = [false,..,false]
D[1..|V|] = [∞,..,∞]
Element[1..|V|] Offer =
                   [null,..,null]
Lock [1.. |V|]  DLock
Lock [1.. |V|]  OfferLock
```

```
relax(v,vd)
 lock(OfferLock[v])
   if (vd < D[v])
     vo = Offer[v]
     if (vo = null)
       Offer[v] = insert(v,vd)
     else
        if (vd < vo.key)
          publishOfferMP(v,vd,vo)
 unlock(OfferLock[v])
```

```
publishOfferMP(v,vd,vo)
  updated = changeKey(vo, vd)
  if (!updated and vd < D[v])
    Offer[v] = insert(v,vd)
```

```
publishOfferNoMP(v,vd)
  Offer[v] = insert(v,vd)
```

```
parallelDijkstra()
 while (!done[tid])
   o = extractMin()
   if (o ≠ null)
     u = o.data
     d = o.key
     lock(DLock[u])
     if(dist < D[u])
       D[u] = d
       explore = true
     else
       explore = false
     unlock(DLock[u])
     if (explore)
       foreach ((u,v) ∈ E)
         vd = d + w(u,v)
         relax(v,vd)
   else
     done[tid] = true
     i = 0
     while (done[i] and i<TNum)
       i =  i + 1
     if(i == TNUM)
        return
     done[tid] = false
```

Figure 6 Parallel versions of Dijkstra's SSSP algorithm: `parallelDijkstra`() is a pseudocode implementation of **ParDijk-MP**. The pseudocode of **ParDijk** can be obtained by replacing the call to `publishOfferMP()` in `relax()` with a call to `publishOfferNoMP()`.

Specifically, the `changeKey`() operation, which is required to update an existing offer $\langle u, d\rangle$ to have distance $d' < d$, is replaced by an `insert`() of a new offer $\langle u, d'\rangle$. As a result, in contrast to the original algorithm, multiple offers for the same node can exist in the queue. Consequently, the parallel algorithm might perform two types of *wasted work*: (1) *Empty work* occurs when a thread dequeues an offer $\langle v, d\rangle$ but then finds that $dist(v) < d$, i.e., that a better offer has already been processed. (2) *Bad work* occurs when a thread updates $dist(v)$ to d, but $dist(v)$ is later updated to some $d' < d$.

In both cases of wasted work, a thread performs an `extractMin`() that would not need to be performed had `changeKey`() been used to update offers in-place, as in the original algorithm. This is particularly detrimental to performance because `extractMin`() operations typically contend for the head of the queue, and the wasted work increases this contention and makes *every* `extractMin`() – wasted or not – more expensive.

Procedure `parallelDijkstra()` shown in Fig. 6 provides the pseudo code of the two parallel versions of Dijkstra's SSSP algorithm that we discuss. The **ParDijk** algorithm is obtained by replacing the call to `publishOfferMP()` in `relax()` with a call to `publishOfferNoMP()`.

The algorithm records its state in several global variables: A boolean array `done` maintains for every thread t a flag `done[`i`]` which records whether the thread found work in the priority queue; an array `D` which records the current estimation of the distance to every node; and an array `Offer` of pointers to offers (elements). Intuitively, `Offer[`u`]` points to the best offer ever made to estimate the distance to node u. The two lock arrays `DLock` and `OfferLock` are use to protect write accesses to arrays `D` and `Offers`, respectively. The locks in `OfferLock` are also used to prevent multiple threads from concurrently changing the priority (distance estimation) to the same node.

When a thread removes an offer $o = (u, d)$ from the queue, it first determines whether it can use it to improve the current distance to u. If this is the case, it updates `D` and turns to exploring u's neighbors, hoping to improve the estimation of their distances too. If the distance to u cannot be shorten, the thread goes back to the queue trying to get more work to do. If the thread managed to improve the distance to u, it explores each of its neighbors v by invoking `relax(v,vd)`. The latter locks `v`'s entry in the `Offer` array, and check whether the new estimation `vd`, is better than the current estimation `D[`v`]` and from the one suggested the best offer so far `Offer[`v`]`. If this is the case, it adds a new offer to the queue. Note that this might lead to node v having multiple offers in the queue.

If the thread does not find work in the queue, i.e., `o` turns out to be *null*, the thread checks if all the other threads have not found work, and if so, terminates.

5.2 ParDijk-MP: A Parallel version of Dijkstra's SSSP Algorithm based on a Concurrent Priority Queue with Mutable Priorities

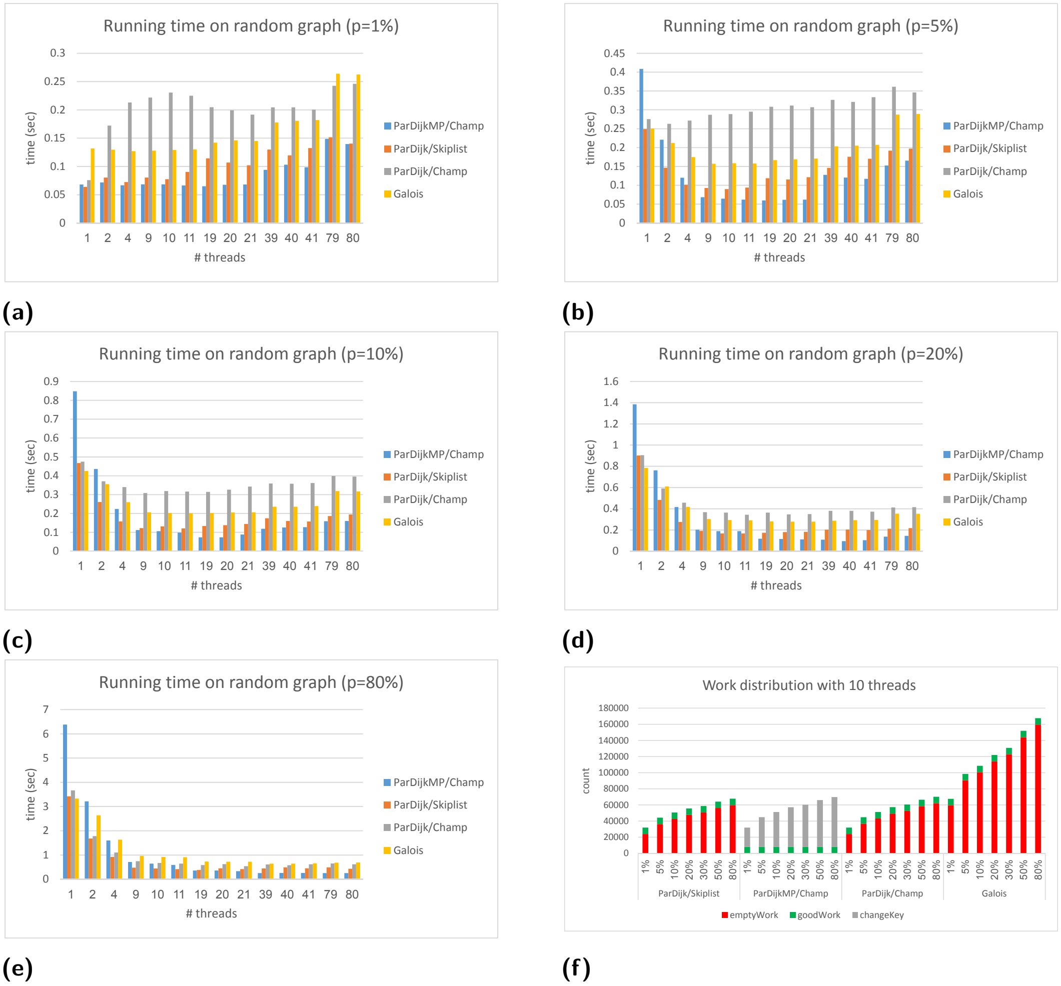
Having a concurrent priority queue which supports a `changeKey`() operation enables updating an existing offer's distance in place, and thus allows parallelizing Dijkstra's algorithm without suffering from wasted work. The change is rather minor: The **ParDijk-MP** algorithm is obtained from procedure `parallelDijkstra()` by keeping the call to `publishOfferMP()` in `relax()`. Note that `publishOfferMP()` checks whether it can update an existing offer in the queue before it tries to insert a new one. This ensures that the queue never contains more than one offer for every node, although a new offer to the same node might be added after the previous offer has been removed.

6 Experimental Evaluation

Our evaluation of Champ focuses on the *overall* performance of the *client application* rather than on the performance of individual core operations. To this end, we used the parallel Dijkstra's algorithms (Section 5) as benchmarks: (1) **ParDijk**, the existing parallel algorithm that may create redundant offers, and (2) **ParDijk-MP**, the version that exploits mutable priorities to update offers in-place. Of these algorithms, only ParDijk can be run with prior priority queues without mutable priorities. We compare Champ to skiplist, a linearizable concurrent priority queue based on a nonblocking skip list, as in the algorithm of Sundell and Tsigas [25].[4] As a performance yardstick, we additionally compare to the parallel SSSP implementation of the Galois [20] graph analytics system. Galois relaxes Dijkstra's algorithm by allowing for both empty work and bad work (see Sec. 5.1). It compensates for the incurred overheads by using a highly-tuned non-linearizable priority queue, which sacrifices exact priority order in exchange for reduced synchronization overhead. We thus use Galois as a representative of the family of *relaxed* non-linearizable priority queues, such as Lotan and Shavit's quiescently consistent algorithm [17] or the SprayList [1].

Experimental setup. We use a Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors. Each processor has 10 2.40 GHz cores, each of which multiplexes 2 hardware threads, for a total of 80 hardware threads. Each core has private write-back L1 and L2 caches; the L3 cache is inclusive and shared by all cores on a processor. The parallel Dijkstra algorithms and priority queues are implemented in Java and run with

[4] Following the methodology of Lindén and Jonsson [15], we implement a singly-linked instead of doubly-linked skip list.



(a) **(b)** **(c)** **(d)** **(e)** **(f)**

Figure 7 SSSP algorithms with different priority queues: Run time (lower is better) and work distribution.

the HotSpot Server JVM, version 1.8.0-25. Galois is implemented in C++; we use the latest version, 2.2.1. All results are averages of 10 runs on an idle machine.

SSSP run time. We measure the running time of each tested algorithm on several input graphs, as we increase the number of threads. Each input is a random graph over 8000 vertices, in which each edge occurs independently with some probability p and a random weight between 1 and 100.[5] Figures 7a–7e depict the results. We observe an overall trend in which all algorithms obtain speedups up to at most 20 threads, but their run time plateaus or increases with more than 20 threads. This is consistent with prior SSSP experiments on identical hardware [1]. We therefore focus our attention on the concurrency levels in which speedups are obtained.

We find that while ParDijk-MP, which leverages Champ's `changeKey()` operation, performs worse than ParDijk/skiplist with few threads, its run time improves as the number

[5] We use the same random weight assignment as Alistarh et al. [1, 14].

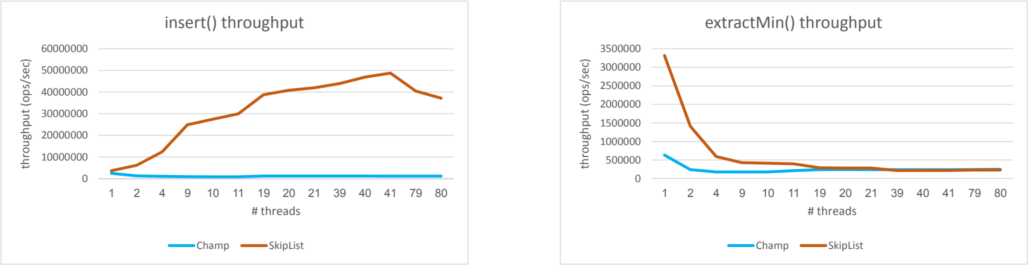


Figure 8 Basic priority queue operations performance: throughput (higher is better) of `insert()` and `extractMin()`.

of threads increases and it eventually outperforms ParDijk/skiplist. On the $p = 1\%$ and $p = 5\%$ graphs, the best run time of ParDijk/skiplist is at 10 threads, and ParDijk-MPis 20% faster than it. Furthermore, the run time of ParDijk-MP plateaus up to 20 threads, whereas ParDijk/skiplist starts deteriorating after 10 threads, making ParDijk-MP $\approx 2\times$ faster than ParDijk/skiplist at 20 threads. On the $p = 10\%$ and $p = 20\%$ graphs, the best run time is at 20 threads, and ParDijk-MPis 60%–80% better than ParDijk/skiplist. On the $p = 80\%$ graph ParDijk-MPoutperforms ParDijk/skiplist only after 20 threads, obtaining a 20% better run time. Similarly, ParDijk-MP outperforms Galois given sufficient parallelism: On the $p = 1\%$ graph ParDijk-MP is consistently about 2× faster, while on the other graphs it is 1.25×–2× slower up to 4 threads, but as more threads are added, its run time becomes 2× better than Galois.

Figure 7f demonstrates the reason for these results, using the 10-thread runs as an example. For each algorithm and input, we classify the work done in each iteration – i.e., for each `extractMin()` – into good work and useless empty work, in which a thread dequeues an outdated offer whose distance is greater than the current distance. (Bad work, in which a thread updated a distance not to its final value, is negligible in all experiments and therefore omitted.) For ParDijk-MP we additionally show the number of `changeKey()` operations performed. As Figure 7f shows, 75%–90% of the work in ParDijk and 90% of the work in Galois is useless. For ParDijk, this corresponds exactly to extraction of outdated offers that in ParDijk-MP are updated in-place using `changeKey()`. In eliminating this useless work, ParDijk-MP with Champ significantly reduces the amount of `extractMin()` operations, which – as we shortly discuss – are the least scalable operations. Note, however, that the gains ParDijk-MP obtains from eliminating the useless work are offset somewhat by Champ's inefficient core operations. We note that we got a similar work distribution when we ran the algorithm with a single thread. This indicates that the wasted work is due to the superfluous insertions and is not an artifact of concurrency.

Turning to ParDijk itself, we find that skiplist outperforms Champ. This occurs because skiplist's `insert()` and `extractMin()` are, respectively, more scalable and more efficient than Champ's. (We discuss this in detail next.) The performance gap between skiplist and Champ shrinks as p increases and the graphs become denser. (For example, at 10 threads, ParDijk's skiplist run time is 3× better than with Champ for $p = 1\%$, 2.16× better for $p = 20\%$ and 1.5× better for $p = 80\%$.) The reason is that as the inputs become denser, threads perform more work – i.e., iterate over more edges – for each offer. Consequently, the priority queue's performance becomes a less significant factor in overall performance: it is accessed less frequently, and thus becomes less contended.

Core operations performance. We study the performance of the core queue operations with microbenchmarks. For `insert`(), we measure the time it takes N threads to concurrently `insert`() 10^6 items ($10^6/N$ each) into the priority queue. For `extractMin`(), we measure the time it takes N threads repeatedly calling `extractMin`() to empty a priority queue of size 10^6. Figure 8 shows the results, reported in terms of the throughput obtained (operations/second). We see that SKIPLIST `insert`() scale well, because insertions to different positions in a skiplist do not need to synchronize with each other. In contrast, every CHAMP `insert`() acquires the heap root lock, to increase the heap size and initiate a `bubbleDown`. As a result, CHAMP insertions suffer from a sequential bottleneck and do not scale. For `extractMin`(), both algorithms do not scale, since both have sequential bottlenecks in extractions. For CHAMP, it is the heap root lock again. For SKIPLIST, it is the atomic (via CAS) update of the pointer to the head of the skiplist.[6] The characteristics of the core operations explain the performance of ParDijk-MP vs. ParDijk: when updating an offer, ParDijk-MP performs a `changeKey`() where ParDijk performs an `insert`(). Both of these are scalable operations, although CHAMP's `changeKey`() may be heavier than a skiplist insertion, as it performs hand-over-hand locking. However, for an offer updated U times, ParDijk performs $U - 1$ extraneous `extractMin`()s that ParDijk-MP/CHAMP avoids. Because `extractMin`() is the most expensive and non-scalable operation, overall ParDijk-MPcomes out ahead.

Sparse vs. dense graphs. In our experiments we used relatively dense graphs. When using sparse graphs like road networks, e.g., Rome99, USA-FLA, USA-NY, and USA-W [6], whose average degree is less than three, we noticed that CHAMP suffers from a slowdown of 2x-15x. We believe that the reason for this behavior is that in these scenarios there is much less wasted work (less than 11% in our experiments). Because there is so little wasted work, the competing algorithms outperform CHAMP due to their faster synchronization, which no longer gets outweighed by the execution on extraneous wasted work.

7 Related Work

Existing concurrent priority queues [1, 2, 11, 15, 16, 17, 23, 24, 25, 26] support `insert`() and `extractMin`() but not `changeKey`, and most of this prior work has focused on designing priority queues with ever more `insert`()/`extractMin`() throughput on synthetic microbenchmarks of random priority queue operations. Researchers have only recently [1, 15, 26] started evaluating new designs on priority queue client applications, such as Dijkstra's algorithm. We are, to the best of our knowledge, the first to step back and approach the question from the client application side, by considering how the `insert`()/`extractMin`() interface restricts the clients, and how to address this problem by extending the priority queue interface.

Our priority queue is based on concurrent heap of Hunt et al. [11], which we extend to support the `changeKey`() operation. We have also changed some of the design choices in [11], to better suit our applications. (See Sec. 4.3). Mound [16] also uses a heap-based structure. It minimizes swapping of heap nodes by employing randomization and storing multiple items in each heap node. It is thus not obvious how to implement `changeKey`() in Mound.

Several concurrent priority queues are based on skiplists [22]. Lotan and Shavit [17] initially proposed such a lock-based priority queue, and Sundell et al. [25] designed a

[6] Note that SKIPLIST's `extractMin`() removes the head (minimum) skip list by first marking it logically deleted and then physically removing it from the list, and any thread that encounters a logically deleted node tries to complete its physical removal before proceeding. This causes further `extractMin`() serialization, on top of the memory contention causes by issuing CASes to the shared head pointer.

nonblocking skiplist-based priority queue. In both algorithms, contention on the head of the skiplist limits the scalability of `extractMin()`. There are two approaches for addressing this bottleneck: One is to improve `extractMin()` synchronization, for example by batching node removals [15] or using combining [2]. Currently this approach does not lead to algorithms that scale beyond 20 threads [2, 15]. A second approach *relaxes* the priority queue correctness guarantees by allowing `extractMin()` to not remove the minimum priority item [1, 23, 26]. Using such algorithms requires reasoning about – and possibly modifying – the application, to make sure it can handle this relaxed behaviors. Note that all these algorithms – relaxed or not – still provide the client with only the limited set of `insert()`/`extractMin()` operations.

8 Conclusions and Future Work

We present and evaluate CHAMP, the first concurrent algorithm for a priority queue with mutable keys. CHAMP is implemented using an array-based binary heap, and consequently its core priority queue operations, `insert()` and `extractMin()`, do not scale as well as in prior designs. Despite this, we show that CHAMP's extended interface improves the performance of parallel versions of Dijkstra's SSSP algorithm, because it saves the client algorithm from *wasting work* when working around the lack of `changeKey()` support in other priority queues. This raises an interesting question for future work: can we efficiently implement mutable priorities in the more scalable skip list-based priority queues without compromising on the scalability of the core operations?

References

1 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue. In *PPoPP*, 2015.

2 Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The Adaptive Priority Queue with Elimination and Combining. In *DISC*, 2014.

3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

4 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

5 E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

6 9th DIMACS implementation challenge. URL: `http://www.dis.uniroma1.it/challenge9/download.shtml`.

7 Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *JACM*, 34(3):596–615, July 1987.

8 Richard M. Fujimoto. Parallel discrete event simulation. *CACM*, 33(10):30–53, 1990. `doi:10.1145/84537.84545`.

9 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

10 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.

11 Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *IPL*, 60(3):151–157, 1996.

12 Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

13 Zvi M. Kedem and Abraham Silberschatz. A characterization of database graphs admitting a simple locking protocol. *Acta Informatica*, 16, 1981.

14 Justin Kopinsky. SprayList SSSP benchmark. `https://github.com/jkopinsky/SprayList/blob/master/sssp.c`, 2015.

15 Jonatan Lindén and Bengt Jonsson. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *OPODIS*, 2013.

16 Yujie Liu and Michael Spear. Mounds: Array-Based Concurrent Priority Queues. In *ICPP*, 2012.

17 Itay Lotan and Nir Shavit. Skiplist-Based Concurrent Priority Queues. In *IPDPS*, 2000.

18 Paul E. McKenney and John D. Slingwine. Read-copy update: using execution history to solve concurrency problems. In *PDCS*, 1998.

19 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 2004.

20 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*, 2013.

21 R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957. `doi:10.1002/j.1538-7305.1957.tb01515.x`.

22 William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *CACM*, 33(6):668–676, June 1990.

23 Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *SPAA*, 2015.

24 Nir Shavit and Asaph Zemach. Scalable Concurrent Priority Queue Algorithms. In *PODC*, 1999.

25 Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *JPDC*, 65(5):609–627, 2005.

26 Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The Lock-free k-LSM Relaxed Priority Queue. In *PPoPP*, 2015.

Maximum Matching for Anonymous Trees with Constant Space per Process*

Ajoy K. Datta[1], Lawrence L. Larmore[2], and Toshimitsu Masuzawa[3]

1 **University of Nevada, Las Vegas, USA**
ajoy.datta@unlv.edu

2 **University of Nevada, Las Vegas, USA**
lawrence.larmore@unlv.edu

3 **Graduate School of Information Science and Technology, Osaka University, Osaka, Japan**
masuzawa@ist.osaka-u.ac.jp

Abstract

We give a silent self-stabilizing protocol for computing a maximum matching in an anonymous network with a tree topology. The round complexity of our protocol is $O(diam)$, where *diam* is the diameter of the network, and the step complexity is $O(n\,diam)$, where n is the number of processes in the network. The working space complexity is $O(1)$ per process, although the output necessarily takes $O(\log\delta)$ space per process, where δ is the degree of that process. To implement parent pointers in constant space, regardless of degree, we use the cyclic Abelian group $\mathbb{Z}_7$.

1998 ACM Subject Classification C.2.4 [Computer-Communication Networks] Distributed Systems, F.2.2 [Analysis of Algorithms and Problem Complexity] Nonnumerical Algorithms and Problems, G.2.2 [Discrete Mathematics] Graph Algorithms

Keywords and phrases anonymous tree,maximum matching,self-stabilization, unfair daemon

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.16

1 Introduction

Self-stabilization [5] is a paradigm for enhancing autonomous adaptability of distributed systems to network dynamics, such as transient faults and topology changes. A self-stabilizing system is guaranteed to regain its intended (or legal) behavior when the system is arbitrarily disturbed. Several fundamental problems have been solved by self-stabilizing algorithms, including leader election, spanning tree construction, mutual exclusion, node/edge coloring, and so forth. Maximum or maximal matching is one of the most investigated of these problems *et al.* [8].

1.1 Related Work

The first self-stabilizing algorithm for *maximal* matching was given by Hsu *et al.* [10]. The algorithm works for arbitrary anonymous networks under the *central daemon*, where no two processes can act simultaneously. Efficiency under the central daemon is usually measured by the number of steps required for convergence to a legitimate configuration. Their algorithm takes $O(n^3)$ steps, where n is the number of processes in the system. This result was improved in [9, 12, 15]. Hedetniemi *et al.* give an algorithm which takes $O(m)$ steps where m is the

* This work was supported in part by JSPS KAKENHI, Grant Number (B)26280022.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 16; pp. 16:1–16:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

number of edges in the system [9]. A synchronous version of the algorithm with round complexity $O(n)$ is given by Goddard *et al.* [6].

A self-stabilizing algorithm for *maximal* matching in arbitrary networks under the *read/write daemon*, where only a single read or write is permitted during an atomic step, was given by Chattopadhyay *et al.* [3]. The algorithm converges in $O(n)$ (asynchronous) rounds, provided all processes have identifiers, and no two processes within distance two have the same identifier. The same paper also gives a randomized self-stabilizing algorithm for assigning the locally distinct identifiers in $O(1)$ expected rounds. With the assumption of distinct identifiers within distance two, a self-stabilizing maximal matching algorithm for arbitrary networks under the *distributed daemon*, where any number of processes can act simultaneously, which converges in $O(n)$ rounds and $O(m)$ steps, was given by Manne *et al.* [13].

A number of self-stabilizing *maximum* matching algorithms have been given. Karaata and Saleh give a self-stabilizing algorithm for anonymous tree networks, under the central daemon, which converges in $O(n^4)$ steps [11]. A self-stabilizing algorithm for tree networks under the read/write daemon, which converges in $O(n^2)$ steps, is given by Blair and Manne [2]. For anonymous bipartite networks, a self-stabilizing algorithms converging in $O(n^2)$ rounds under the central daemon is given by Chattopadhyay *et al.* [3].

A maximal matching is defined to be *1-maximal* if the size of matching cannot be increased by replacing one edge in the matching with two other edges. Any 1-maximal matching is a $\frac{2}{3}$-approximation of the maximum matching, while an arbitrary maximal matching is only a $\frac{1}{2}$-approximation to maximum.

A self-stabilizing 1-maximal matching algorithm, under the central daemon, for anonymous trees, and for rings with length not divisible by three, which converges in $O(n^4)$ rounds, is given by Goddard *et al.* [7]. The same paper also shows that there is no self-stabilizing 1-maximal matching algorithm for anonymous rings of length a multiple of three. These results are generalized by Asada and Inoue, who give a self-stabilizing 1-maximal matching algorithm for any anonymous network under the central daemon, provided that network has no cycle of length divisible by three, which converges in $O(m)$ steps [1].

For non-anonymous arbitrary networks, a self-stabilizing 1-maximal matching algorithm under the distributed daemon is given by Manne *et al.* [14]. Their algorithm converges in $O(n^2)$ rounds, but requires exponentially many steps in the worst case.

1.2 Contribution

We use of *virtual pointers*, in which we indicate a parental relation between neighboring processes by assigning each process a label of finite size, that is, written with finitely many bits, where the parental relation is a function of the labels of the two processes. In this paper, each label, which we call *level*, is a member of the cyclic Abelian group $\mathbb{Z}_7$, and x is a parent of y if the difference of the two values of *level* is in a certain range. We are able to maintain constant space per process by eliminating the standard parent pointers, which require $O(\log\delta)$ space per process, where δ is its degree.

Match builds a spanning tree, or a spanning structure consisting of two trees, in the network. Match then uses a bottom-up dynamic program to choose a maximum matching for the network. The working space complexity of Match is $O(1)$ per process, regardless of degree, the round complexity is $O(\mathit{diam})$, where *diam* is the diameter of the network, and the step complexity is $O(n\ \mathit{diam})$. Our use of $\mathbb{Z}_7$ is similar to the use of $\mathbb{Z}_5$ for virtual pointers in [4]. The output must use $O(\log\delta)$ space per process, where δ is the degree of that process.

1.3 Outline

In Section 2, we give some basic definitions, and describe our model of computation. In Section 4, we formally define Match. In Section 5, we prove that Match is correct. In Section 6 we prove that Match takes $O(diam)$ rounds, while in Section 7, we prove that Match takes $O(n\, diam)$ steps. Section 8 concludes the paper.

2 Preliminaries

We say that a network is *undirected* if the edges have no specified orientation, *i.e.,* that the edge $\{x, y\}$ is the same as the edge $\{y, x\}$. We say that a network is *anonymous* if the processes have no identifiers. We say that a network is a *tree* if it is connected and has no cycles. The algorithm we give in this paper assumes an undirected anonymous tree.

2.1 Model of Computation

We use the shared memory model of computation [5], meaning that each process can read its own registers and those of its neighbors, and can change only its own registers. A distributed algorithm consists of a program for each process, and that program consists of a finite set of *actions*. Each action for a process x has a *guard*, which is a predicate (*i.e.*,, Boolean function) on the registers of x and its neighbors, together with a *statement*, which is simply the assignment, or reassignment, of one or more registers of x. If the guard of an action of x is TRUE, then we say that action is *enabled*, and we say x is enabled if at least one of its actions is enabled.

We assume the *unfair distributed daemon*. If at least one process is enabled, the daemon *selects* at least one of these enabled processes. Each selected process executes one of its enabled actions, and that concludes one step. We describe the daemon as *unfair* because an enabled process need never be selected, unless it becomes the only enabled process.

We will assume that there is a set $\mathcal{F}$ of configurations which we call the *legitimate* configurations. (We call the remaining configurations *illegitimate*.) $\mathcal{F}$ satisfies two conditions:

Closure: No action can change a legitimate configuration to an illegitimate configuration.

Correctness: Each legitimate configuration satisfies the output conditions of the problem.

A configuration is *final* if no process is enabled at that configuration. An algorithm is *silent* if every computation ends at a final configuration.

3 Overview of Match

We are given an anonymous unoriented network G with a tree topology. The first phase of Match is to build a rooted spanning tree for G, or possibly two rooted trees joined at the roots which together span G. During the second phase (which actually begins before the first phase is finished) we use the tree, or trees, to define a maximum matching.

At any given configuration of Match, every neighbor of a process x is either a *child* of x, a *parent* of x, or a *peer* of x. From a possibly chaotic initial configuration, Match organizes G into one or two trees; if there are two trees, the roots are peers of each other.

The maximum matching itself is then constructed by a bottom-up wave of the tree (or trees). Each process is assigned the Boolean label 0 or 1, which we call *flag*, as follows. Leaves are assigned 0. A process is assigned 1 if it has a child labeled 0, otherwise 0. At the end, every process labeled 1 is matched with one of its children labeled 0, while two roots are matched with each other if both are labeled 0.

Virtual Pointers. It is typical to define rooted trees by using parent pointers. But we need to maintain $O(1)$ space complexity at each process, regardless of degree. For that reason, we express parent/child relations using *virtual pointers.* These virtual pointers are defined by storing a single variable, $x.level$ at each process. The value of $x.level$ will be an element of $\mathbb{Z}_7$, the cyclic Abelian group of order 7, and such requires only three bits to store.

More formally, if $i, j \in \mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$, we say that $i < j$, and $j > i$, if $j - i \in \{1, 2, 3\}$, where addition and subtraction are in the Abelian group, *e.g.*, $2 < 3$, $2 < 4$, $2 < 5$, and $6 < 2$. This relation are not transitive, since $1 < 3 < 5$, but $5 < 1$. If y is a neighbor of x in the network we say that y is a child of x if $y.level > x.level$. Similarly, y is a parent of x if x is a child of y, *i.e.,* $y.level < x.level$, while y is a peer of x if $y.level = x.level$. Thus, to change the parent/child/peer relationships, we simply change the values of *level*.

How Virtual Pointers are Used. An algorithm which uses local addresses for pointers never has to change its parent pointer unless it changes its parent. That simple rule does not hold for $\mathbb{Z}_7$-virtual pointers. We say that $parent(x) = y$ if $y.level < x.level$. A problem then arises if it is necessary to change the level of either x or y. If $x.level = 2 + y.level$, the "ideal" parent/child relation, either $x.level$ or $y.level$ may be either incremented or decremented without changing the parent/child relationship. However, if $x.level = 1 + y.level$, decrementation of $x.level$ would cause x and y to become peers; similarly, if $x.level = 3 + y.level$, incrementation of $x.level$ would cause the parent/child relation to be reversed: x would become the parent of y. For this reason, whenever $parent(x) = y$ and $x.level \neq 2 + y.level$, x tries to either increment or decrement its level to restore the ideal. Until that ideal is restored, y is not enabled to execute an action which changes its level.

Think of the link between parent and child as a spring which has an ideal length (namely 2) but whose length can be stretched to 3 or compressed to 1. After that distortion, the spring tries to restore its length to the ideal 2.

We use the group $\mathbb{Z}_7$ because there is no smaller group which allows the same flexibility in both directions, namely both compression and stretching of the parent/child link.

3.1 Approach

In order to describe how Match creates a tree, or trees, we need to introduce an abstract function $Level(x)$. *Level* is an integer function, and has the following properties:

1. The standard projection $\mathbb{Z} \to \mathbb{Z}_7$ maps $Level(x)$ to $x.level$. For example, if $Level(x) = 11$ then $x.level = 4$. *i.e.,*
2. If $y \in N(x)$, then $|Level(y) - Level(x)| \leqslant 3$.
3. $Level(x)$ increments (decrements) at each step at which $x.level$ increments (decrements).

We also introduce the abstract function $Min\_Level = \min\{Level(x) : x \in G\}$. *Level* can be initialized arbitrarily; however in Lemma 6 below, we prove that Min_Level is constant, and thus we can assume, without loss of generality, that $Min\_Level = 0$, which implies that, during the computation, all values of *Level* are non-negative. Eventually, $Level(x) = 2\,d(x)$ for each process, where $d(x)$ is the distance, through G, from x to the root of the tree, or the nearer root if there are two trees.

4 Formal Definition of Match

Variables and functions of Match. We use the dot notation, such as $x.var$, for a variable of x, while we use the normal functional notation such as $Func(x)$, for a function of x.

1. $x.level \in \mathbb{Z}_7$. This is the variable that allows us to define virtual parent pointers using constant space per process: x is the parent of y if $x.level < y.level$.
2. $Chldrn\,(x) = \{y \in N(x) : y.level > x.level\}$, the *children* of x, where $N(x)$ is the set of neighbors of x.
3. $Peers\,(x) = \{y \in N(x) : y.level = x.level\}$, the *peers* of x, those neighbors at the same level as x in the spanning tree.
4. $Prnts\,(x) = \{y \in N(x) : y.level < x.level\}$, the *parents* of x, those neighbors above x in the spanning tree.
5. $Children\_Ok\,(x) \equiv \forall y \in Chldrn\,(x)(y.level = 2 + x.level)$, Boolean. This means that all children of x are in the “optimum" position, meaning two levels below x.
6. $\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}, \text{IV}, \text{V}\}$, which we define below. At a final configuration, if there is one root, it has Class I, while if there are two roots, they each have Class II; while in both cases, all other processes have Class III.
7. $x.flag$, Boolean. This variable is used in the bottom-up protocol which determines the matched pairs. Except for the possibility of two roots being matched, all matched pairs consist of processes with opposite values of *flag*.
   The values of *flag* are computed using a bottom-up dynamic program, and those values define a maximum matching, once the structure of the rooted spanning tree, or trees, is finalized.
8. $x.partner \in N(x) \cup \{\perp\}$, the partner that x is matched to. If $x.partner = \perp$, then x is unmatched. This variable, which is the output of the protocol, takes $O(\log\delta)$ space, where δ is the degree of x.

Classes of Processes. At each configuration, each process belongs to one of five Classes; its Class is defined by its number of parents and peers, as given by the table below. The Class of a process is arguably its most important property.

# parents	# peers	Class
0	0	I
0	1	II
1	0	III
0	$\geqslant 2$	IV
1	$\geqslant 1$	V
$\geqslant 2$	arbitrary	V

We write $peer(x)$ for the sole peer of a process in Class II, and we write $parent\,(x)$ for the sole parent of a process in Class III.

For any process x, let T_x be the subtree rooted at x, namely the set consisting of x, its children, its children's children, *etc.*. We define a process x to be *regular* if $\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}\}$ and all descendants of x have Class III. At a final configuration, all processes are regular.

We now define additional variables and functions computable by a process.

9. $x.rglr$, Boolean. This variable means that x currently “believes” that it is regular.
   We define a process to be *strongly regular* if it is regular and $y.rglr$ for all $y \in T_x$.
10. $Rglr\,(x) \equiv (\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}\}) \wedge (\forall y \in Chldrn\,(x) y.rglr)$, Boolean. This predicate is used to update $x.rglr$.
11. $Flag(x) = \begin{cases} \text{TRUE if } (\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}\}) \wedge (\exists y \in Chldrn\,(x)\, y.flag = \text{FALSE}) \\ \text{FALSE otherwise} \end{cases}$
    This predicate is used to update $x.flag$.

12. $\mathit{Partner}(x) \in N(x) \cup \{\perp\}$, the process that x should be matched with. If $x.\mathit{flag} = \textsc{true}$, then $\mathit{Partner}(x)$ is some child of x whose flag is $\textsc{false}$, if any, while if $\mathrm{Class}(x) = \mathrm{II}$, $x.\mathit{flag} = \textsc{false}$, and $\mathit{peer}(x).\mathit{flag} = \textsc{false}$, then $\mathit{Partner}(x) = \mathit{peer}(x)$. If $\mathrm{Class}(x) = \mathrm{III}$, $x.\mathit{flag} = \textsc{false}$, and $\mathit{parent}(x).\mathit{partner} = x$, then $\mathit{Partner}(x) = \mathit{parent}(x)$. In all other cases, $\mathit{Partner}(x) = \perp$.

The variable $x.\mathit{partner}$, and the corresponding function $\mathit{Partner}(x)$, use $O(\log \delta)$ space per process, but are used only for output. All intermediate computations use $O(1)$ space per process, hence we say that MATCH uses constant *working space* per process.

Regular, *Rglr*, and *rglr*. We have used the word "regular," or contractions thereof, in three different ways. *Regularity* is an abstract property, not computable by any process during the execution of MATCH; $x.\mathit{rglr}$ is a working estimate of the regularity of x.

Code of MATCH

We now present the protocol MATCH for an arbitrary process x, in program form. We define a process x to be *enabled* if execution of the protocol by x results in the change of at least one variable of x. At each step, the daemon selects an arbitrary non-empty set of enabled processes; if there are no enabled processes, the configuration is final.

```
1: if $x.rglr \neq Rglr(x)$ then
2:    $x.rglr \leftarrow Rglr(x)$
3: else if $Children\_Ok(x)$ then
4:    if $(\mathrm{Class}(x) = \mathrm{II}) \wedge x.rglr \wedge \neg peer(x).rglr$ then
5:       $x.level \leftarrow x.level + 1$
6:    else if $(\mathrm{Class}(x) = \mathrm{III}) \wedge x.rglr \wedge (x.level = parent(x).level + 1)$ then
7:       $x.level \leftarrow x.level + 1$
8:    else if $(\mathrm{Class}(x) = \mathrm{III}) \wedge x.rglr \wedge (x.level = parent(x).level + 3)$ then
9:       $x.level \leftarrow x.level - 1$
10:   else if $(\mathrm{Class}(x) = \mathrm{V}) \wedge \neg x.rglr \wedge (y \in Peers(x) \cup Prnts(x) \implies \neg y.rglr)$ then
11:      $x.level \leftarrow x.level - 1$
12:   end if
13: end if
14: if $x.rglr \wedge (x.flag \neq Flag(x))$ then
15:    $x.flag \leftarrow Flag(x)$
16: else if $x.rglr \wedge x.partner \neq Partner(x)$ then
17:    $x.partner \leftarrow Partner(x)$
18: end if
```

Notation. In our discussion, we will say that a process executes an *action* if it executes one of the lines of the code which changes one of its variables, *i.e.,* Line 2, 5, 7, 9, 11, 15, or 17. Note that a process is enabled to execute at most one of those actions per step.

We say that a process executes a *level action* if it executes a line that changes its level, namely Line 5, 7, 9, or 11.

Intuitive Explanation of the Actions

Line 5 increases the level of a regular process, and changes its Class from II to III, provided its peer is not regular; its peer then becomes its parent. In Figure 1, **b** executes the action at

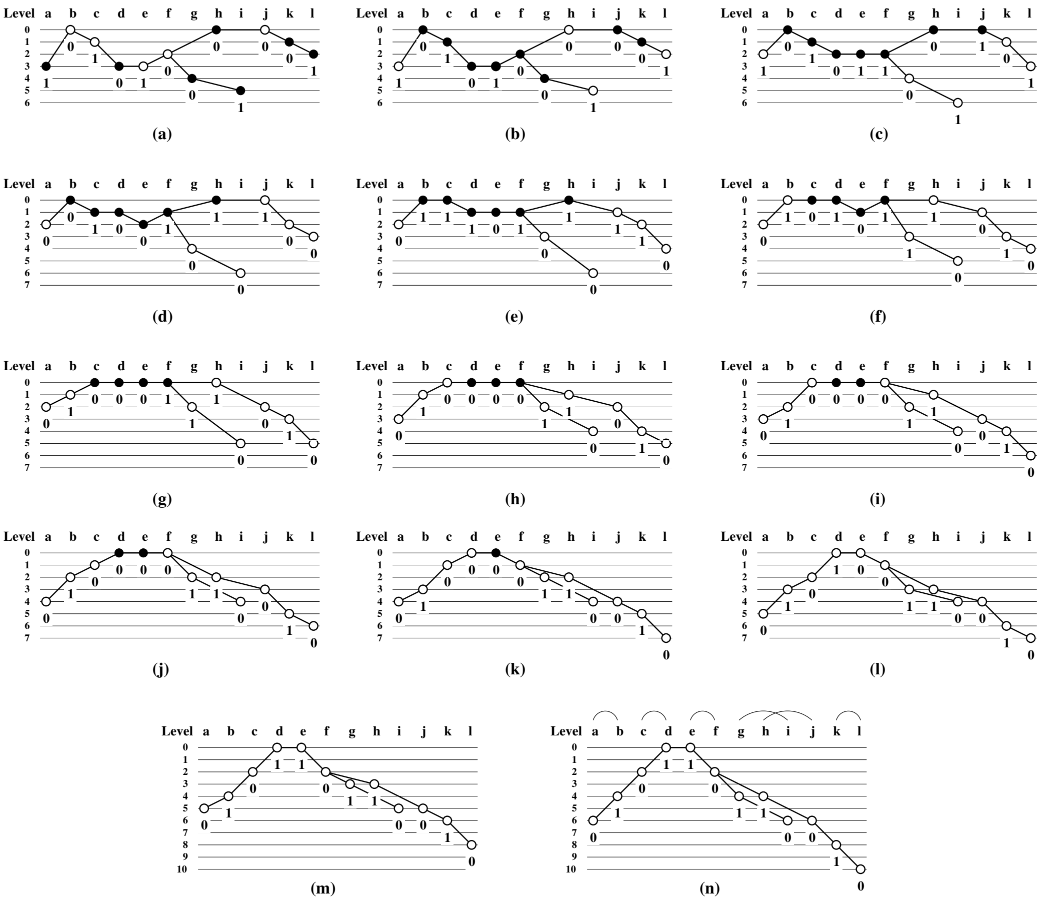


Figure 1 Example computation of Match, where G consists of twelve processes, named $\mathbf{a}, \mathbf{b}, \ldots \mathbf{k}, \mathbf{l}$. Values of *flag* are indicated under the processes. Process x is shown by an open dot if $x.rglr$, otherwise a solid dot. Configuration (a) is "arbitrary." Processes **d** and **e** are computed to be roots, though neither has level zero in the initial configuration. We assume the synchronous daemon, *i.e.,* at each step every enabled process executes. Several steps are skipped between (m) and the final configuration (n). To avoid clutter, matching actions, *i.e.,* executions of Line 17, are not shown except in the last figure. The matched pairs are $\{\mathbf{a}, \mathbf{b}\}$, $\{\mathbf{c}, \mathbf{d}\}$, $\{\mathbf{e}, \mathbf{f}\}$, $\{\mathbf{g}, \mathbf{i}\}$, $\{\mathbf{h}, \mathbf{j}\}$, and $\{\mathbf{k}, \mathbf{l}\}$.

(f), **h** executes the action at (g), **c** executes the action at (i), and **f** executes the action at (j).

Lines 7 and 9 adjust the levels of regular processes of Class III without changing their virtual pointers; in those cases, a process x adjusts the difference between its level and that of its parent from 1 or 3 to 2; execution of those lines has no effect on the parent/child/peer structure of the network. There are many examples of those actions illustrated in Figure 1, such as **k** at (c). In fact, at each step from (b) to (m), at least two processes of Class III execute a level action.

Of the actions shown in Figure 1, Line 15 has the lowest priority, meaning that the value of $x.\mathit{flag}$ cannot change if x executes Line 2 or any level action. All $x.\mathit{flag}$ have reached their final values, as shown in (m), before the level actions are finished. The level actions are the last to be completed in the example, and the final configuration is shown in (n).

Execution of Line 11 of the code decreases the level of process of Class V. We do not allow a process x to execute that action unless $y.rglr = \text{FALSE}$ for all $y \in \mathit{Peers}(x) \cup \mathit{Prnts}(x)$.

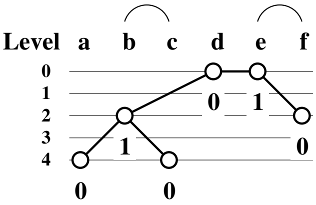


Figure 2 An example final configuration of Match, where G is a tree consisting of six processes, named $\mathbf{a}, \mathbf{b}, \ldots \mathbf{f}$. Values of *flag* are indicated under the processes. The matched pairs are $\{\mathbf{b}, \mathbf{c}\}$ and $\{\mathbf{e}, \mathbf{f}\}$, while **a** and **d** remain unmatched.

There are a number of examples of this action in Figure 1, such as processes **d** and **e** at (b), **d** and **f** at (c), **c** and **e** at (d).

The Kernel. Let K be the set of all processes of Classes I, II, IV, and V, together with all irregular processes of Class III. We call K the *kernel*. The purpose of Line 11 is to squeeze K into the 0^{th} level. Once that has happened, as shown in Figure 1(g), all irregular processes have Class IV, and K is a tree subnetwork (actually merely a chain in Figure 1) of G whose leaves all have Class II. In the remaining steps, that subnetwork will be decremented as its leaves change Class from II to III.

4.1 Top Level Summary of Match

At the top level, Match executes the following level actions.

1. All processes in the K move to the 0^{th} level, executing Line 11. K will be a tree.
2. The leaves of that tree will all have Class II. Each Class II process will increase its level by executing Line 5, deleting itself from the kernel, and leaving the 0^{th} level.
3. Eventually, the kernel will consist of either one or two processes at the 0^{th} level. If one process, it will be the root. If two, and $rglr = \text{TRUE}$ for both, they will be co-roots.
4. As all these level actions are proceeding, regular processes will continually execute Lines 7 and 9, continually adjusting differences between the levels of parents and children to make the links ideal, so that the other level actions will be enabled.
5. The Flag and Matching actions, Lines 15 and 17, implement a simple dynamic program that assigns a flag value for each process, and then chooses a maximum matching in a rooted tree. A process of flag 1 always has at least one child of flag 0, and matches with it. A process of flag 0 cannot match with any of its children. In the example shown in Figure 1, there are no unmatched processes. However, if a process has more than one child of flag 0, one of those children will remain unmatched, as indicated in Figure 2 below; in that figure **b** has flag 1 and its children both have flag 0; thus, one of those children must remain unmatched.
6. At the very end of the dynamic program, if there are two roots, they match with each other if they both have flag 0. A root with flag 1 matches with one of its children, as *per* the above rule. Thus, if one root has flag 1 and the other 0, the root with flag 0 remains unmatched, as shown in Figure 2 below.

4.2 Legitimate Configurations of Match

A legitimate configuration consists of one or two trees, whose root, or roots, are at level zero. All parental links are ideal, meaning that the difference in levels between parent and child is two in every case. Furthermore, in a legitimate configuration, $x.rglr = \text{TRUE}$ for all x. It follows that there is either one process of Class I or two neighboring processes of Class II, and all other processes are of Class III. In addition, and $x.flag = Flag(x)$ for all x, and all processes have matched with their final partners, or remain unmatched.

Every final configuration is legitimate, as we prove in Lemma 1. However, a legitimate configuration may not be final. For example, Figure 1(m) shows a legitimate but non-final configuration. The legitimate configuration is not unique. There is no general polynomial bound on the number of distinct maximum matchings of a tree graph.

5 Correctness

Correctness of Match follows from the following two statements:

1. Every final configuration of Match is legitimate, as we prove in Lemma 1 below,
2. There is no infinite computation of Match, as we prove in Lemma 12 below.

▶ **Lemma 1.** *Any final configuration of* Match *is legitimate.*

Proof. Suppose that the current configuration of Match is final, *i.e.*, no process is enabled to execute any action of the code.

▶ Claim 2. $x.rglr = \text{TRUE}$ for any process x.

Proof of Claim 2. Suppose not. Let $L = \max\{Level(x) : \neg x.rglr\}$, and let x be a process of level L such that $x.rglr = \text{FALSE}$. Thus $y.rglr = \text{TRUE}$ for all $y \in Chldrn(x)$. Hence, if the Class of x is I, II, or III, $Rglr(x) = \text{TRUE}$, contradiction.

▶ Claim 3. If Class $(x) = \text{V}$ then x is enabled.

Proof of Claim 3. If $y \in Prnts(x)$ and $y.rglr$, then y is enabled to execute Line 2, contradiction. If $y \in Peers(x)$, then Class $(y) = \text{IV}$, hence $y.rglr = \text{FALSE}$, because otherwise y would be enabled. Thus, x is enabled. ◀

The only remaining possibility is that Class $(x) = \text{IV}$. Since each process at level L has Class IV, and each has at least two peers also at level L, the subgraph of processes at level L must contain a cycle, contradiction. This completes the proof of Claim 2. ◀

By Claim 2 and since $x.rglr = Rglr(x)$ for all x, there are no processes of Class IV or V. Since any process of Class III has a parent, any process at Level zero must have Class I or II.

▶ Claim 4. If $Level(x) = 0$ and Class $(x) = \text{I}$, then all processes other than x have Class III and are descendants of x.

Proof of Claim 4. Suppose $y \neq x$ is a process, and y is not a descendant of x. Let σ be the unique path through G from x to y, and let z be the process of maximum level in σ which is closest to x. If $z \neq y$, then z has Class V, contradiction. Thus, $z = y$, and y is a descendant of x. Since all processes are descendants of x, all other processes must have Class III. ◀

▶ Claim 5. If there is no process of Class I at level zero, then there are two processes of Class II at level zero which are peers of each other, and every other process has Class III.

Proof of Claim 5. There must be a process x of Class II at level zero. Let $y = peer(x)$. Let z be any other process, and let σ be the unique path through G from z to x. By an argument similar to that in the proof of Claim 4, z is a descendant of y if σ passes through y, and otherwise is a descendant of x. In either case Class $(z) = \text{III}$. ◀

The lemma follows from Claims 4 and 5. ◀

▶ **Lemma 6.** *Min_Level does not change during any computation of* MATCH*.*

Proof. By contradiction. Suppose *Min_Level* decreases during a step. Then there is some process x such that $\mathit{Level}(x) = \mathit{Min\_Level}$ and $\mathit{Level}(x)$ decreases at the next step. Since no process has a level less than $\mathit{Level}(x)$, we know that Class (x) is neither III nor V. However, the guards of the actions do not permit $x.\mathit{level}$ to decrease if x belongs to any other Class. Thus, *Min_Level* does not decrease.

Now, suppose *Min_Level* increases during the step. For any process x such that $\mathit{Level}(x) = M = \mathit{Min\_Level}$ before the step, $\mathit{Level}(x)$ must increase, hence Class (x) must be either II or III. But x has no parent, hence Class $(x) =$ II, which implies that x has a peer. Since $\mathit{peer}(x).\mathit{rglr} =$ FALSE, $\mathit{Level}(\mathit{peer}(x))$ cannot decrease, and thus there is still a process whose level is M after the step, contradiction. ◀

▶ Remark 7. If a process x is regular at some step, then x is regular at all subsequent steps.

No Computational Cycle. In this paragraph, we assume that Ξ is a non-trivial computational cycle of MATCH. Our goal is to prove that Ξ cannot exist.

▶ **Lemma 8.** *No process changes its regularity during* Ξ*.*

Proof. By Remark 7, a process does not change from regular to irregular. Since every step of Ξ must be reversible, no process can change from irregular to regular during Ξ. ◀

▶ **Lemma 9.** *If* $x.\mathit{rglr} =$ TRUE *at any step of* Ξ*, then* x *is regular.*

Proof. By contradiction. Suppose the lemma is false. Let S be the set of ordered triples (x, t, L) such that the process x is irregular at the t^{th} step of Ξ, and that $x.\mathit{rglr} =$ TRUE and $\mathit{Level}(x) = L$ at that step. Pick such a triple (x, t, L) such that L is maximum over all members of S. Without loss of generality, x executes an action at step t.

By Lemma 8, x is irregular at step $t-1$. If Class $(x) =$ V at step $t-1$, then $x.\mathit{rglr} =$ FALSE at step t, contradiction. Otherwise, there is some $y \in \mathit{Chldrn}^{t-1}(x)$ such that y is irregular at step $t-1$. Let $L' = \mathit{Level}(y)$ at step $t-1$. If $y.\mathit{rglr} =$ FALSE at step $t-1$, then $x.\mathit{rglr} =$ FALSE at step t, contradiction. Thus $y.\mathit{rglr} =$ TRUE at step $t-1$, hence $(y, t-1, L') \in S$, and $L' > L$, which contradicts the maximality of L. ◀

▶ **Lemma 10.** *During* Ξ*, no irregular process changes level.*

Proof. By contradiction. Suppose x is irregular and changes its level during Ξ. Since Ξ is a cycle, $x.\mathit{level}$ must both increase and decrease during Ξ. By Lemma 9, $x.\mathit{rglr} =$ FALSE during Ξ, and thus by the definitions of the actions $x.\mathit{level}$ can only decrease, contradiction. ◀

▶ **Lemma 11.** *During* Ξ*, no regular process changes level.*

Proof. We first observe that no process of Class III can change to any other Class. Suppose x is a regular process. If Class $(x) =$ I, it cannot change its level. If Class $(x) =$ II, then x cannot change its level, because its Class would change to III, and that step would be irreversible. Thus Class $(x) =$ III.

The statement of the lemma is proven by induction on level. If $\mathit{Level}(x) = 0$, then Class $(x) \neq$ III. If $\mathit{Level}(x) > 0$, let $y = \mathit{parent}(x)$. By either Lemma 10 or the inductive hypothesis, depending on whether y is irregular or regular, y does not change level during Ξ. Thus, x can change level at most once during Ξ, and that change is irreversible. Since Ξ is a cycle, x does not change level at all. ◀

▶ **Lemma 12.** *The computation of* Match *is acyclic.*

Proof. Suppose Ξ is a cycle of computation of Match. By Lemmas 8, 10, and 11, no process executes Line 2 nor any level action during Ξ. Given that regularity and levels are fixed, a process can execute neither Line 15 nor Line 17 infinitely often, and thus Ξ cannot exist. Hence Match is acyclic. ◀

▶ **Theorem 13.** Match *is correct.*

Proof. Correctness follows immediately from Lemmas 1 and 12. ◀

6 Round Complexity

Monus Notation. The operator *monus,* written "$\dot{-}$" on non-negative integers is a variant of subtraction, but never yields a negative. Formally, $x \dot{-} y = \max\{x - y, 0\}$. For example, $5 \dot{-} 3 = 2$, while $3 \dot{-} 5 = 0$. Monus is left-associative and has the same precedence as addition and subtraction. Note that $i \dot{-} j \dot{-} k = i \dot{-} (j + k)$.

We define a sequence of potentials. Let SR be the set of strongly regular processes.

1. $\pi(x) = \begin{cases} -1 \text{ if } (\text{Class}\,(x) = \text{III}) \wedge (x \in SR) \wedge (x.level = 2 + parent(x).level) \\ 1 \text{ otherwise} \end{cases}$
2. $\theta(x) = \max\{0, \pi(x) + \max\{\theta(y) : y \in Chldrn(x)\}\}$
3. $\mu(x) = \begin{cases} 0 \text{ if } Chldrn(x) = \varnothing \\ \max\{\theta(y) : y \in Chldrn(x)\} \text{ otherwise} \end{cases}$
4. $d(x) =$ the distance through G from x to r where r is the root that will eventually be computed by Match. If two roots are computed, take r to be the one closer to x.
5. $\psi(x) = d(x) + 2\,Level(x)$
6. $\Psi = \max\{\psi(x) : x \in K \cup I\}$ where $I = \{x : \neg x.rglr\}$.
7. $\alpha(x) = \begin{cases} 1 \text{ if } (\text{Class}\,(x) = \text{I}) \wedge (x \in I) \\ 1 \text{ if } (\text{Class}\,(x) = \text{II}) \wedge (x \in I \vee peer(x) \notin I) \\ 2 \text{ if } (\text{Class}\,(x) = \text{V}) \wedge (x \notin I) \\ 1 \text{ if } (\text{Class}\,(x) = \text{V}) \wedge (x \in I) \wedge (Peers(x) \cup Prnts(x) \nsubseteq I) \\ 0 \text{ otherwise} \end{cases}$
8. $\phi(x) = 4\,\psi(x) + \mu(x) + \alpha(x)$.
9. $\Phi = \max\{\phi(x) : x \in K \cup I\}$.

The Potential Φ Decreases. In a sequence of lemmas, we prove that Φ decreases. Henceforth, let $X = \{x \in K \cup I : \phi(x) = \Phi\}$.

▶ Remark 14. (a) For any $x \in K \cup I$, $\phi(x)$ does not increase. (b) Φ does not increase.

▶ **Lemma 15.** *If $x \in X$ and $\mu(x) = 0$, then either $\phi(x)$ decreases or $x \notin K \cup I$ during the next round.*

Proof. All descendants of x are strongly regular, since otherwise $\phi(x)$ would not be maximal.
Case 1. Class $(\boldsymbol{x}) =$ **I.** If $x \in I$, then x is enabled to execute Line 2, and will do so within one round, decreasing $\alpha(x)$, hence decreasing $\phi(x)$. If x is the sole process at level 0, then $x = r$, hence $\Phi = 0$, contradiction. Otherwise, there is a path σ from x to some process at level 0. This path must contain a process y of Class V whose level is greater than that of x, implying that $\phi(y) > \phi(x)$, contradiction.
Case 2. Class $(\boldsymbol{x}) =$ **II.** If $\alpha(x) = 0$, then x will execute Line 5 within one round, becoming completely regular, and hence leaving X. If $\alpha(x) = 1$, either x will execute Line 2 or $peer(x)$ will execute Line 2, or both. After that execution $\alpha(x) = 0$.

Case 3. Class $(\boldsymbol{x}) = \mathbf{III.}$ Then $x \in I$, and x will execute Line 2 within one round, after which $x \notin K \cup I$.
Case 4. Class $(\boldsymbol{x}) = \mathbf{IV.}$ There must be some $y \in \mathit{Peers}(x)$ such that $d(y) > d(x)$. Thus $\phi(y) > \phi(x)$, contradiction.
Case 5. Class $(\boldsymbol{x}) = \mathbf{V.}$ If $\alpha(x) = 2$, then x will execute Line line: regular within one round, after which $\alpha(x) \leqslant 1$. If $\alpha(x) = 1$, then every member of $\mathit{Peers}(x) \cup \mathit{Prnts}(x)$ will execute Line 2, after which $\alpha(x) = 0$. If $\alpha(x) = 0$, x will execute Line 11 within one round, decreasing the value of $\phi(x)$. ◀

Level Actions of Class III Processes. If x is a Class III process, we say that x has *type* 0 if $x.\mathit{level} = 2 + \mathit{parent}(x).\mathit{level}$. Otherwise, we say that x has type 1.

▶ Remark 16. If x is regular, then x is enabled to execute a level action if and only if x has type 1 and all children of x have type 0.

▶ **Lemma 17.** *No two neighboring Class III processes are simultaneously enabled to execute a level action.*

Proof. If two Class III processes are neighbors, one must be the parent of the other. The result then follows immediately from Remark 16. ◀

▶ **Lemma 18.** *If x is a strongly regular process which is enabled to execute a level action, then x will execute that action within the next round.*

Proof. All we need to show is that x cannot be neutralized before it acts. Since x has type 1, $\mathit{parent}(x)$ cannot change its level, and since all children of x have type 0, they also cannot change level. ◀

Chains not Trees. We analyze the evolution of the function θ only for the case that every subtree of regular processes is a chain. Our logic is that the evolution of $\theta(x)$ is determined by the worst case behavior of any chain of T_x. (Henceforth, when we say “chain" of processes we shall always mean a chain that ends at a leaf.)

Bit String Representation of Chains. We replace a chain σ of regular processes by a bit string $w(\sigma)$, obtained by replacing each process by its type. We index the symbols of a string starting from the right; for example, if $w = 01$, then $w_1 = 1$ and $w_2 = 0$. The i^{th} *suffix* of w, $S_i(w)$, is defined to be the suffix of w starting at w_i, *i.e.*, $S_i(w) = w_i w_{i-1} \cdots w_1$. We define θ recursively for both a string, and a symbol within a string, as follows.

1. $\theta(\varepsilon) = 0$, where ε is the empty string.
2. $\theta(0w) = \theta(w) \dot{-} 1$
3. $\theta(1w) = \theta(w) + 1$.
4. $\theta(w, i) = \theta(S_i(w))$.

▶ **Lemma 19** (Monotonicity of θ)**.** *Let w be a bit string.*
(a) *If any 1 in w is replaced by 0, $\theta(w)$ does not increase.*
(b) *If any collection of substrings 10 in w are each replaced by 01, $\theta(w)$ does not increase.*

Proof. Let w and w' be strings such that $\theta(w) \geqslant \theta(w')$. Then

▶ Claim 20. $\theta(0w) \geqslant \theta(0w')$.

▶ Claim 21. $\theta(1w) \geqslant \theta(1w')$.

▶ Claim 22. $\theta(1w) > \theta(0w)$.

▶ Claim 23. $\theta(10w) \geqslant \theta(01w')$.

Claims 20, 21, and 22 follow trivially from the definition of θ. By Claims 20 and 21, we have $\theta(10w) = \theta(w) \dot{-} 1 + 1 \geqslant \theta(w) = \theta(01w) \geqslant \theta(01w')$, proving IV. We have (a) by recursion on $|w|$, using Claims 20, 21, and 22, and (b) by recursion on $|w|$, using 20, 21, and 23. ◀

▶ **Lemma 24.** *If w is a bit string, $\theta(w) > 0$, and w' is obtained from w by replacing each substring 10 by 01, and $w'_1 \leftarrow 0$, then $\theta(w') < \theta(w)$.*

Proof. The proof is by induction. The inductive step consists of a number of cases, each characterized by the values of w_i, w_{i-1}, and w'_{i-1}. There are special cases for $i = 1$. We leave the details of this proof for the full paper. ◀

▶ **Lemma 25.** *If $\Phi > 0$, then Φ decreases during the next round.*

If it were not for the requirement that a process x can only change its level if *Children_Ok* (x) is true, we would be able to prove that Φ decreases every round. What is true is that, at every configuration where $I \neq \varnothing$, every process whose value of ϕ is maximum is enabled, provided that the (annoying) condition *Children_Ok* holds.

The term $\mu(x)$ provides the needed correction. It measures the number of rounds needed before *Children_Ok* (x) becomes true. The coefficient of 4 is needed because of the time needed to ensure μ is again zero on some process of maximum ϕ.

Proof. Let $\Phi > 0$. Let $X = \{x \in K \cup I : \phi(x) = \Phi\}$.

Let $x \in X$. We need to show that $\phi(x)$ decreases within one round. If $\mu(x) = 0$, then $4\psi(x) + \alpha(x)$ decreases, by Lemma 15.

▶ Claim 26. If $\mu(x) = 0$ and x executes a level action, then within one round, $\mu(x)$ does not increase by more than 2.

Sketch of Proof of Claim 26. If x is enabled to execute a level action, and σ is a maximal chain of regular processes ending with a child of x, then $\theta(\sigma) = 0$, which implies that the top member of σ, a child of x, is of type 0, meaning that the bit string $w = w(\sigma)$ starts with 0. When x executes the level action, that 0 is replaced by either 1 or 11, depending on the action. In the worst case, $\theta(w)$ is increased by 3, by Lemmas 19 and 24. ◀

We omit the remaining details of the proof of the lemma. ◀

The Strongly Regular Case. We now consider the case that all processes are strongly regular.

▶ **Lemma 27.** *If all processes are strongly regular, then within $O(diam)$ rounds, all processes are of type 0.*

Proof. We use Lemma 24 and monotonicity of θ, namely Lemma 19. Again simplifying to the case of a chain of length h, the worst case of a strongly regular chain is where each process is of type 1, and a chain is represented by the bit string $111\ldots111$ of length h, where $h \leqslant diam$. Within h rounds the bit string will consist of alternating zeros and ones, and within another h rounds, it will be all zeros. By monotonicity, any other string will become all zeros in the same number of rounds, or fewer. ◀

▶ **Theorem 28.** *The round complexity of* Match *is $O(diam)$.*

Proof Sketch: Each of the terms that makes up the potential Φ is $O(diam)$, and thus by Lemma 25, Φ decreases until $I = \emptyset$ and K consists only of the one process of Class I or the two processes of Class II, within $O(diam)$ rounds.

The configuration now satisfies the conditions of the strongly regular case, and within $O(diam)$ additional rounds, every process is at its final level.

Within $O(diam)$ additional rounds, the flag values converge and processes are matched. The total round complexity of Match is thus $O(diam)$. ◀

7 Step Complexity

In this section, we sketch the proof that Match has step complexity $O(n\,diam)$. We can assume, without loss of generality, that only one process executes at any given step. Henceforth in this section, we assume that each step consists of the execution of one process.

"Star" Notation. We use a star superscript on a variable or function to indicate the value of that variable or function at the end of the current computation. For example, we write $parent^*(x)$ for the value of $parent(x)$ at the final configuration of the computation, and T_x^* for the subtree rooted at x in the final configuration.

We say that a configuration of Match is *aligned* if $parent(x) = y$ implies $parent^*(x) = y$; otherwise we say the configuration is *chaotic*.

We will give the complete proof of the step complexity of Match in the full paper. In this extended abstract, we outline the proof in the aligned case. By Lemma 29, alignment is a closed property.

▶ **Lemma 29.** . *Alignment is a closed property.*

Proof. We only sketch the proof. The configuration is chaotic if and only if there is an *inverted pair*, which we define to be neighboring processes x, y such that $parent(x) = y$ and $parent^*(y) = x$. The only action that can create an inverted pair is Line 11, but careful inspection of this action shows that it can only create an inverted pair if there is already an inverted pair. Thus, an aligned configuration can never become chaotic. ◀

7.1 Regularity Actions

In this subsection, we prove that the total number of executions of Line 2 during a computation of Match which starts at an aligned configuration is $O(n\,diam)$.

Let $Chldrn^*(x) = \{y \in N(x) : parent^*(y) = x\}$, the *eventual children* of x.

▶ **Lemma 30.** *During a computation of* Match *starting from an aligned configuration, the total number of executions of Line 2 is $O(n\,diam)$.*

Proof. We first introduce some potentials.

1. $\varrho(x) \equiv (x.rglr \neq Rglr(x))$, Boolean, meaning that x is enabled to execute Line 2. We write 0 or 1 for the values of ϱ.
2. $\tau(x) = \begin{cases} 1 \text{ if } x.rglr \wedge (\text{Class}(x) = \text{II}) \\ 0 \end{cases}$
3. $\omega(x) = \varrho(x) + \sum \omega(y) : y \in Chldrn^*(x)$
4. $\Omega = \sum \{\omega(x) + \tau(x) : x \in G\}$

▶ Claim 31. $\omega(x) \leqslant |T_x^*|$.

Proof of Claim 31. By induction on $height(T^*_x)$. If $height(T^*_x) = 0$, meaning x is a leaf of the final tree, then $\omega(x) = \varrho(x) \leqslant 1$. Suppose $height(T^*_x) > 0$. By the inductive hypothesis, $\omega(x) = \varrho(x) + \sum\{\omega(y) : y \in Chldrn^*(x)\} \leqslant 1 + \sum\{|T^*_y| : y \in Chldrn^*(x)\} = |T^*_x|$. ◀

Recall that $d(x)$ is the distance, through G, from x to the nearest root r. Clearly, $d(x) \leqslant diam$. For any d, Let $\Omega_d = \sum\{\omega(x) : d(x) = d\}$.

▶ Claim 32. $\Omega_d \leqslant n$ for all d.

Proof of Claim 32. T^*_x and T^*_y are disjoint if $d(x) = d(y)$. Thus $\Omega_d = \sum\{|T_x| : d(x) = d\} \leqslant n$. ◀

▶ Claim 33. $\Omega = O(n\,diam)$.

Proof of Claim 33. By Claim 32 $\Omega = \sum_{d=0}^{diam} \Omega_d + \sum\{\tau(x) : x \in G\} \leqslant n\,(diam + 2)$ ◀

▶ Claim 34. Ω does not increase, and Ω decreases at each step where some process executes Line 2.

Proof of Claim 34. Consider the execution of one action by a process x. The only actions that have an effect on the potentials ϱ, τ, ω, and Ω are those listed in the table below. In each case that x executes Line 2, Ω decreases, while in the one other case listed, an action of Line 5, the value of Ω does not increase. The table below summarizes the effect of each of those actions on the potentials, where Δ indicates the increase of a quantity at the step.

	$\Delta\varrho(x)$	$\Delta\omega(x)$	$\Delta\varrho(y)$	$\Delta\omega(y)$	$\Delta\tau(x)$	$\Delta\Omega$
Class (x) = I; x executes Line 2.	-1	-1			0	-1
Class (x) = II; x executes Line 2; $y = peer(x)$.	-1	-1	0	-1	1	-1
Class (x) = II; x executes Line 5; $y = peer(x)$.	0	0	$\leqslant 1$	$\leqslant 1$	-1	$\leqslant 0$
Class (x) = III; x executes Line 2; $y = parent(x)$.	-1	-1	$\leqslant 1$	$\leqslant 0$	0	$\leqslant -1$

The other actions have no effect on any of the potentials listed above. ◀

The lemma follows from Claims 33 and 34. ◀

▶ **Lemma 35.** *During a computation of* Match *starting from an aligned configuration, the number of steps at which some process executes Line 15 is* $O(n\,diam)$.

We skip the proof of Lemma 35, which is almost identical to the proof of Lemma30. We can show that the step complexity of a computation starting from an aligned computation consists of $O(n\,diam)$ steps, using Lemmas 30 and 35, as well as additional lemmas which we postpone to the full paper. In the full paper, we will prove the step complexity of Match.

8 Conclusion

We have given a self-stabilizing algorithm, under the unfair distributed daemon, for finding a maximum matching of the processes of an anonymous network with a tree topology. Our algorithm runs in $O(diam)$ rounds and $O(n\,diam)$ steps, and needs only $O(1)$ working space per process, that is, space required for intermediate computations.

References

1 Y. Asada and M. Inoue. A silent self-stabilizing algorithm for 1-maximal matching in anonymous networks. In *Proc. WALCOM 2015*, pages 187–198, 2015.

2 J.R.S. Blair and F. Manne. Efficient self-stabilizing algorithms for tree networks. In *Proc. ICDCS 2003*, pages 20–26, 2003.

3 S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing. In *Proc. PODC 2002*, pages 290–297, 2002.

4 A. K. Datta and L. L. Larmore. Leader election and centers and medians in tree networks. In *Proc. SSS 2013*, pages 113–132, 2013.

5 E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

6 W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proc. IPDPS 2003*, page 162, 2003.

7 W. Goddard, S.T. Hedetniemi, and Z. Shi. An anonymous self-stabilizing algorithm for 1-maximal matching in trees. In *Proc. PDPTA 2006*, pages 797–803, 2006.

8 N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.

9 S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Maximal matching stabilizes in time $O(m)$. *Information Processing Letters*, 80(5):221–223, 2001.

10 S.C. Hsu and S.T. Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.

11 M.H. Karaata and K.A. Saleh. A distributed self-stabilizing algorithm for finding maximum matching. *Computer Systems Science and Engineering*, 15(3):175–180, 2000.

12 M. Kimoto, T. Tsuchiya, and T. Kikuno. The time complexity of Hsu and Huang's self-stabilizing maximal matching algorithm. *IEICE Trans. Infrmation and Systems*, E93-D(10):2850–2853, 2010.

13 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. In *Proc. SIROCCO 2007*, pages 96–108, 2007.

14 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science*, 412(4):5515–5526, 2011.

15 G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.

Atomic Snapshots from Small Registers

Leqi Zhu[1] and Faith Ellen[2]

1 University of Toronto, Toronto, Canada
2 University of Toronto, Toronto, Canada

Abstract

Existing n-process implementations of atomic snapshots from registers use large registers. We consider the problem of implementing an m-component snapshot from small, $\Theta(\log n)$-bit registers. A natural solution is to consider simulating the large registers. Doing so straightforwardly can significantly increase the step complexity. We introduce the notion of an *interruptible* read and show how it can reduce the step complexity of simulating the large registers in the snapshot of Afek et al. [1]. In particular, we show how to modify a recent large register simulation [2] to support interruptible reads. Using this modified simulation, the step complexity of UPDATE and SCAN changes from $\Theta(nm)$ to $\Theta(nm + mw)$, instead of $\Theta(nmw)$, if each component of the snapshot consists of $\Theta(w \log n)$ bits. We also show how to modify a limited-use snapshot [4] to use small registers when the number of UPDATE operations is in $n^{O(1)}$. In this case, we change the step complexity of UPDATE from $\Theta((\log n)^3)$ to $O(w + (\log n)^2 \log m)$ and the step complexity of SCAN from $\Theta(\log n)$ to $O(mw + \log n)$.

1998 ACM Subject Classification E.1 Distributed data structures

Keywords and phrases atomic snapshot, limited-use snapshot, small registers, simulation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.17

1 Introduction

Atomic snapshots give processes the ability to obtain a consistent view of shared memory through a SCAN operation, even when other processes are concurrently performing UPDATE operations to the memory. This allows programmers to reason about the concurrency in the system in a higher-level manner and can greatly simplify development and verification of concurrent programs. In their seminal paper on atomic snapshots, Afek et al. [1] cite many applications. In the same paper, they presented an n-process, m-component snapshot implementation from registers (i.e., using only READ and WRITE) with $\Theta(nm)$ step complexity for both SCAN and UPDATE.

A well-known concern with this implementation, and indeed all known snapshot implementations from registers, is the assumption that the system provides registers large enough to store the result of a SCAN. As the number of components or the size of each component of the snapshot grows, this assumption becomes less and less practical. We consider the problem of implementing snapshots shared by n processes from $\Theta(\log n)$-bit registers. We call such registers *words*. It is customary to use registers with $\Omega(\log n)$ bits, so they can store process identifiers.

A natural solution is to consider simulating the large registers. Recently, Aghazadeh, Golab, and Woelfel [2] showed how to simulate a $\Theta(w \log n)$-bit register from words with optimal step complexity, $\Theta(w)$, for READ and WRITE. Straightforwardly applying their register simulation to the snapshot implementation by Afek et al. significantly increases the step complexity from $\Theta(nm)$ to $\Theta(nmw)$, if each component of the snapshot consists of $\Theta(w \log n)$ bits.

Most of these extra steps come from (unnecessarily) reading the embedded scans and value fields contained in each of the large registers during the double-collects. However, it is possible to determine if the embedded scans and values are needed by reading only $\Theta(\log n)$ bits from each of the registers. Motivated by this, we introduce the notion of an *interruptible* read. This means that a process can read part of the large register's data, pause the simulated READ to perform other operations, and then return to read more of the data, so that the entire read appears to occur at the same time. The large register simulation of Aghazadeh, Golab, and Woelfel can be modified to have interruptible reads. Applying this modified simulation to the snapshot of Afek et al. reduces the step complexity of UPDATE and SCAN from $\Theta(nmw)$ to $\Theta(nm + mw)$.

Recently, Aspnes, Attiya, Censor-Hillel, and Ellen [4] showed that, if the number of UPDATE operations, b, is in $n^{O(1)}$, then it is possible to implement a b-limited-use m-component snapshot from $\Theta(nmw \log n)$-bit registers with step complexity $\Theta((\log n)^3)$ for UPDATE and step complexity $\Theta(\log n)$ for SCAN. Simulating the large registers from words increases the step complexity of UPDATE to $\Theta(n^2mw)$ and the step complexity of SCAN to $\Theta(nmw)$, even with interruptible reads.

We show how to directly modify their implementation to use $\Theta(\log n)$-bit registers while only slightly increasing the step complexity: $O(w + (\log n)^2 \log m)$ for UPDATE and $O(mw + \log n)$ for SCAN. The idea is similar to interruptible reads. Instead of directly returning a view, a SCAN returns an index into a sequence of views, whose length is proportional to the number of UPDATE operations that have been performed. We call this an *implicit* SCAN. The view may then be examined by using this index as input to a VIEW operation. We call snapshots implemented in this way *implicit*. We provide a simple recursive construction of an implicit snapshot from two implicit snapshots with fewer components. Instead of representing a view directly, we represent it implicitly, by a pair of indices into the sequences of views of these two smaller implicit snapshots. The actual view can be recovered recursively. This also allows us to also implement a *partial* scan [5], in which only c of the components are queried, with step complexity $O(c(w + \log m) + \log n)$.

2 Model and Preliminaries

We consider an asynchronous shared memory system with n processes which communicate using shared $\Theta(\log n)$-bit (multi-writer) registers. We call these registers *words*. We assume that processes may fail at any time by crashing.

An *execution* in this system is an alternating sequence of *configurations* and *events* $C_0, e_1, C_1, \ldots$. Each event e_i (or, *step*) is either a READ or WRITE operation on a shared register and each configuration C_i consists of the contents of every register and the state of each process after event e_i is applied to configuration C_{i-1}. For any two events a and b in an execution, we write $a \to b$ to mean that a precedes b in the execution.

An *implementation* of a shared object in this system provides a representation of the object using words and an algorithm for each type of operation supported by the object and for each process sharing the object. We only consider *wait-free* implementations, where each operation invoked by a non-faulty process is guaranteed to be completed within a finite number of its own steps. The *step complexity* of an operation O in an implementation is the maximum, over all possible executions, of the number of steps taken by any process to finish an instance of O that it invoked.

Given an execution, the *execution interval* of an operation is the portion of the execution which begins with the first step in the operation and ends with the last step in the operation.

An implementation is *linearizable* [8] if, for every execution, we can choose a *linearization point* in the execution interval of each operation such that operations appear to occur instantaneously at their linearization points.

An m-component *atomic snapshot* (or simply *snapshot*) has two operations, SCAN and UPDATE(j, v). The SCAN operation returns an instantaneous *view* of the components, as if all m components were read in a single atomic step. The UPDATE(j, v) operation updates component j to have value v and returns nothing. For $b \geq 1$, we say a snapshot is *b-limited-use* if it supports at most b UPDATE operations in any execution.

A *max register* has two operations, READ-MAX and WRITE-MAX(v). The READ-MAX operation returns the largest value written thus far, while WRITE-MAX(v) adds a number v to the set of values written and returns nothing. For $b \geq 1$, we say a max register is *b-bounded* if its values are restricted to $\{0, \ldots, b-1\}$. Aspnes, Attiya, and Censor-Hillel [3] showed that:

▶ **Theorem 1.** *There is a b-bounded max register implementation with step complexity $\Theta(\log b)$ which uses only 1-bit registers.*

A 2-component *max array* has two operations, MAX-SCAN and MAX-UPDATE(j, v). Each component behaves like a *max* register. The MAX-SCAN operation returns an instantaneous view of the two components, as if both components had READ-MAX performed on them in a single atomic step. The MAX-UPDATE(j, v) operation updates component j as if it had performed WRITE-MAX(v) to it and returns nothing. For $b_1, b_2 \geq 1$, we say a 2-component max array is *(b_1, b_2)-bounded* if the values of its first component are restricted to $\{0, \ldots, b_1 - 1\}$ and the values of its second component are restricted $\{0, \ldots, b_2 - 1\}$. Aspnes, Attiya, Censor-Hillel, and Ellen [4] showed that:

▶ **Theorem 2.** *There is a (b_1, b_2)-bounded 2-component max array implementation with step complexity $\Theta(\log b_1 \log b_2)$ which uses only 1-bit registers.*

3 Unlimited-use snapshot from small registers

We show how to obtain an m-component snapshot implementation from words with step complexity $\Theta(nm + mw)$ for SCAN and UPDATE, if each component of the snapshot consists of $\Theta(w \log n)$ bits.

Our approach is to simulate the large registers in the m-component snapshot by Afek et al. [1] from words. In their implementation, a SCAN performs $\Theta(n)$ collects on an array of m registers, each containing $\Theta(w \log n)$ bits. The large register simulation by Aghazadeh, Golab, and Woelfel [2] has $\Theta(w)$ step complexity for READ and WRITE of a $\Theta(w \log n)$-bit register. Thus, if we directly apply their simulation, the step complexity of SCAN and UPDATE (which contains an embedded SCAN) becomes $\Theta(nmw)$ instead of $\Theta(nm)$.

We observe that not all the bits read during the collects are needed. Indeed, in all but the last collect, only $\Theta(\log n)$ bits from each of the m registers end up being used. Furthermore, after reading only these bits, it is possible to determine which additional bits need to be read. Motivated by this, we introduce the notion of an *interruptible* read which, intuitively, allows a process to reserve a copy of the value in the large register (BEGIN-IREAD), read particular words in the copy (READ-WORD), and then return the memory for reuse (END-IREAD). In general, this is useful for algorithms in which a process can read only a small fraction of the bits in a large register to determine if it needs to read the rest of the bits.

In Section 3.1, we formally define an interruptible read. In Section 3.2, we explain how to modify the simulation by Aghazadeh et al. to implement BEGIN-IREAD, READ-WORD,

p_1	W(00)			W(11)			
p_2		$\text{BI} = t_1$	$\text{RW}(t_1, 1) = 0$		$\text{BI} = t_2$	$\text{RW}(t_2, 1) = 1$	$\text{RW}(t_2, 0) = 0$

Figure 1 Example of an execution using interruptible reads.

and END-IREAD in constant time. In Section 3.3, we carefully apply this modified simulation with interruptible reads to the standard snapshot of Afek et al. to obtain an m-component snapshot with SCAN and UPDATE step complexity $\Theta(nm + mw)$. Finally, in Section 3.4, we describe some other, faster snapshot implementations and why it is difficult to modify them to use words while maintaining their step complexity.

3.1 Interruptible reads

Formally, a simulation of a $\Theta(w \log n)$-bit register from words supports *interruptible* reads if it implements 3 operations: BEGIN-IREAD$_p$, READ-WORD$_p$, and END-IREAD$_p$, for each process p. BEGIN-IREAD$_p$ takes no arguments and returns a pointer to a block of w words which represents the current value of the large register. END-IREAD$_p$ takes a pointer returned by a BEGIN-IREAD$_p$ operation and returns nothing. READ-WORD$_p$ takes a pointer t returned by a BEGIN-IREAD$_p$ operation and an integer j. It returns the value of the j'th word in the block of memory pointed to by t. Between a BEGIN-IREAD$_p$ that returns a pointer t and the next occurrence of END-IREAD$_p(t)$, the memory pointed to by t is not changed. We say that a process p has an *active* interruptible read at the end of an execution if the execution contains a BEGIN-IREAD$_p$ operation that returns some pointer t which is not followed by a corresponding END-IREAD$_p(t)$ operation.

For example, we can use interruptible reads to implement a normal READ by obtaining a pointer t via BEGIN-IREAD$_p$, concatenating the values returned by READ-WORD$_p(t, j)$ for $j = 1, \ldots, w$ into a single value v, releasing the memory pointed to by t via END-IREAD$_p(t)$, and then returning v.

Figure 1 gives an example of an execution involving 2 processes, p_1 and p_2, using interruptible reads. It features a 2-bit register being simulated by 1-bit registers. At the start, p_1 writes 00, denoted by W(00). Then p_2 begins an interruptible read and obtain a pointer t_1. This is denoted by $\text{BI} = t_1$. Next, p_2 reads the first word (in this case, a bit) being pointed to by t_1, which has value 0. We denote this by $\text{RW}(t_1, 1) = 0$. Now p_1 writes 11. Then p_2 begins another interruptible read to obtain a pointer t_2 and reads the first word being pointed to by t_2, which has value 1. Finally, when p_2 reads the second word pointed to by t_1, it is still 0. At the end of this execution, p_2 has 2 active interruptible reads.

3.2 A large register simulation supporting fast interruptible reads

Not all large register simulations support fast interruptible reads. For instance, Peterson [10] showed how to simulate a large *single-writer* $\Theta(w \log n)$-bit register from *single-writer* $\Theta(\log n)$-bit registers. His simulation represents a large register by collections of $\Theta(w)$ words, called *buffers*. The writer alternately writes to two of these buffers. A switch bit indicates which of these two buffers was most recently written to. The writer flips the switch bit after completing a sequence of $\Theta(w)$ writes to one of these buffers. Ideally, the readers would read from one buffer while the writer writes to the other. However, processes may fall asleep for a long time. Using handshakes, the writer can detect if a reader is concurrent with its WRITE. In this case, it also writes the current value of the register to the reader's designated copy buffer. The reader performs collects on both of the main buffers as well as its own copy buffer. By checking the switch and handshake bits, the reader returns the value of a buffer that was not being written to while it was being read. Implementing fast interruptible reads is difficult in this case because the reader cannot quickly determine which pointer BEGIN-IREAD should return.

The simulation by Aghazadeh et al. [2] can easily be modified to support a polynomial (in n) number of active interruptible reads per process. Like Peterson's simulation, they use buffers. Writers have a pool of buffers to which they may write and there is a pointer to the most recently written buffer. To READ, a reader reads the pointer to the most recently written buffer and announces this pointer. The algorithm guarantees that an announced buffer will not be modified by any writer. Since a writer may miss this announcement, there is also a mechanism for a writer to pass hints to the reader about alternate buffers which have been written to in the meantime, from which it is safe to read. These hints can be read by the reader in a constant number of steps. The algorithm guarantees that, until the reader acknowledges a hint, no writer is allowed to modify the buffers mentioned in the hint. The reader acknowledges a hint when it will no longer read from the buffer to which it points. This takes a constant number of steps. It is possible, but not necessary, for the reader to clear its initial announcement.

To implement interruptible reads, we break this READ operation into pieces. In particular, BEGIN-IREAD is the portion of the READ that determines the buffer to be read. It returns a pointer to that buffer. READ-WORD simply reads the appropriate word from the buffer. By the correctness of the simulation, the words may be read in any order. Finally, END-IREAD is be the portion of the READ that acknowledges hints. Note that each of these operations take a constant number of steps.

This simulation assumes that each process can only have one operation active at a time. To support $c \in n^{O(1)}$ active interruptible reads of the same large register per process, we need to increase the size of the buffer pool for each writer by a factor of c. Then the size of each pointer increases by $\lceil \log_2 c \rceil = \Theta(\log n)$ bits.

3.3 Application to Afek et al.

We consider the m-component snapshot implementation of Afek et al. [1]. Suppose that each component of the snapshot consists of $\Theta(w \log n)$ bits. The implementation uses binary registers, $q_{i,j}$ and $q'_{i,j}$, for $i, j \in \{0, 1, \ldots, n-1\}$, which are the *handshaking* bits, $\Theta(mw \log n)$-bit registers, $view_1, \ldots, view_n$, which store views, and $\Theta(w \log n)$-bit registers, $R_1, \ldots, R_m$, each of which stores the current value of the component, a process identifier, and a *toggle* bit.

A SCAN operation by process p_i consists of a loop, each iteration of which (1) collects $q_{1,i}, \ldots, q_{n,i}$, (2) writes to $q'_{i,1}, \ldots, q'_{i,n}$, (3) collects $R_1, \ldots, R_m$ twice, and (4) collects $q_{1,i}, \ldots, q_{n,i}$ again. The iteration is *successful* if the handshaking bits read in both collects of $q_{1,i}, \ldots, q_{n,i}$ are the same and the process identifiers and toggle bits read in both collects of $R_1, \ldots, R_m$ are the same. In this case, (5) the current value of the components read from the second collect of $R_1, \ldots, R_m$ are returned. Otherwise, the algorithms (6) checks (by examining the previously read handshaking bits, process identifiers, and toggle bits) if some process p_j has performed at least one complete UPDATE since the start of the SCAN and, if so, (7) reads and returns $view_j$. They prove that there can be at most $O(n)$ iterations of the loop. An UPDATE operation by process p_i consists of a collect of $q'_{1,i}, \ldots, q'_{n,i}$, writes to $q_{i,j}$, for $j \in \{0, 1, \ldots, n-1\}$, an embedded SCAN, a write to $view_i$, and a write to R_c, for some $c \in \{1, \ldots, m\}$.

Steps (1), (2), and (4) have $\Theta(n)$ step complexity. If we directly apply the simulation of Aghazadeh et al. [2], the step complexity of step (3) will be $\Theta(mw)$ and the step complexity of the SCAN is $\Theta(nmw)$. We can improve this to $\Theta(m)$ using interruptible reads by reading only the words containing the process identifiers and toggle bits from $R_1, \ldots, R_m$ in both collects. We can end the interruptible reads started during the first collect immediately after the first collect is finished. If the iteration is successful, then we read the current value of each component and end each interruptible read started during the second collect. If the iteration is unsuccessful, then we end each interruptible read started during the second collect without reading the current value of each component. We note that each process has at most one active interruptible read on each register at any time. Steps (5) and (7) have step complexity $\Theta(mw)$. Step (6) consists of only local operations. Since steps (5) and (7) are performed just before returning, they contribute $\Theta(mw)$ steps to the total. Steps (1) to (4) occur $\Theta(n)$ times in the worst case. Overall, the step complexity of SCAN is $\Theta(nm + mw)$.

▶ **Theorem 3.** *For $k \in \Omega(\log n)$, there is an m-component snapshot implementation from k-bit registers with step complexity $\Theta(nm + mw)$ for SCAN and UPDATE, if each component of the snapshot consists of $\Theta(wk)$ bits.*

3.4 Other snapshots

There are snapshots with better step complexity. Attiya and Rachman [7] implemented a *single-writer* snapshot from *single-writer* registers with $\Theta(n \log n)$ step complexity. However, they perform too many reads on large registers. In particular, they perform $\Theta(n \log n)$ reads of $\Theta(nw \log n)$-bit registers, if each component of the snapshot consists of $\Theta(w \log n)$ bits. Moreover, they always use the entire value obtained from each read, so interruptible reads are not helpful. This results in a step complexity of $\Omega(n^2 w \log n)$ using any large register simulation. Inoue and Chen [9] showed how to implement lattice agreement from multi-writer registers with $\Theta(n)$ step complexity. Attiya, Herlihy, and Rachman showed how to implement a single-writer snapshot from lattice agreement [6]. This implies an implementation of a single-writer snapshot from multi-writer registers with $\Theta(n)$ step complexity. Unfortunately, the implementation of Attiya, Herlihy, and Rachman uses unbounded size registers. It is unclear whether it is possible to modify these implementations to use small registers while maintaining their step complexity.

4 Limited-use snapshot from small registers

An *implicit snapshot* object is like a regular snapshot object except that a SCAN operation is separated into two parts, an ISCAN operation and a VIEW operation. Intuitively, the ISCAN operation is where the actual SCAN occurs. It returns a pointer to a view, which may then be read via the VIEW operation. To facilitate our implementation of a *partial* SCAN [5], a VIEW operation takes a range of components as input and returns the values of the components in that range. We formalize this as follows.

An ISCAN operation S returns a value $t(S) \geq 0$, which we call the *index* of S. We require that, for any two ISCAN operations S_1 and S_2, $t(S_1) < t(S_2)$ if and only if S_1 is linearized before S_2 and there is at least one UPDATE linearized between them. We also require that an ISCAN operation S has $t(S) = 0$ if and only if no UPDATE operation is linearized before it.

Given an ISCAN operation S, we define the *view at index* $t(S)$ to be the m-component vector whose c'th component contains the value of the last UPDATE operation to component c linearized before S, or the initial value $\perp$, if no such operation exists, for all $c \in \{1, \ldots, m\}$. This is well-defined since, by our requirement on the indices returned by ISCANs, there can be no UPDATE operations linearized between two ISCAN operations returning the same index. The VIEW(t, i, j, V) operation, takes as input an index t returned by a previously completed ISCAN operation, integers $1 \leq i \leq j \leq m$, and an output array $V[1..j - i + 1]$. It writes components i through j of the view at index $t(S)$ into entries 1 through $j - i + 1$ of V. A VIEW could simply return an array containing the values of components i to j. However, since we recursively build implicit snapshots from implicit snapshots with fewer components in Section 4.2, it is more efficient to copy values to one array versus repeatedly creating arrays and concatenating them.

We note that an implicit snapshot object can implement a regular snapshot object by substituting the SCAN operation with VIEW(ISCAN$(), 1, m, V[1..m])$. Furthermore, an implicit snapshot can implement a *partial* SCAN [5] on a set of disjoint component ranges $(i_1, j_1), \ldots, (i_r, j_r)$ by performing ISCAN and then running VIEW$(t, i_k, j_k, V[c])$ for all $k \in \{1, \ldots, r\}$, where t is the index returned by the initial ISCAN.

4.1 A 1-component limited-use implicit snapshot implementation

We can implement a b-limited-use 1-component single-writer implicit snapshot using an array A of b single-writer registers and a single-writer register index. To UPDATE component 1 to v, the writer increments a local counter t, writes v to A$[t]$, and then writes t to index. An ISCAN reads index and returns it. VIEW$(t, 1, 1, V)$ reads A$[t]$ and writes this value to $V[1]$.

To extend this implementation to multiple writers, we change index to be a $(bn + 1)$-bounded max register. Furthermore, instead of incrementing a local counter, a writer performs index.*read-max* to determine the current index t, chooses an index $t' > t$ that no other writer will choose, writes the value v to A$[t']$, and then performs index.*write-max*(t'). An ISCAN consists of performing index.*read-max* and returning the resulting value. The VIEW operation is unchanged. Always choosing t' to be the smallest integer larger than t which is congruent to the writer's process identifier modulo n ensures that different indices are chosen by different writers and they are all bounded above by bn. See Algorithm 1 for the pseudocode.

Algorithm 1 A b-limited-use 1-component implicit snapshot object.

```
1: procedure UPDATE$(1, u)$
2:     $t \leftarrow$ index.read-max()
3:     $t' \leftarrow \min\{j : j > t$ and $j \equiv i \bmod n\}$        ▷ code for process $p_i$
4:     A$[t']$.write$(u)$
5:     index.write-max$(t')$
6: procedure ISCAN
7:     return index.read-max()
8: procedure VIEW$(t, 1, 1, V[1..1])$
9:     $V[1] \leftarrow$ A$[t]$.read()
```

Recall that, from Theorem 1, there is a linearizable implementation of a bounded max register from binary registers. Thus, we will assume that all operations on index are atomic and treat them as steps in the executions we consider.

For every UPDATE operation U, let $t(U)$ be the value t' used as the argument of the index.*write-max* that U performed on line 5.

▶ **Lemma 4.** *For all $t \geq 1$, there is at most one UPDATE operation U with $t(U) = t$. No UPDATE operation U has $t(U) = 0$.*

Proof. By line 3, if $t \equiv i \bmod n$, then only process p_i can choose t. Since p_i performs index.*write-max*(t) on line 5, the values returned to p_i from index.*read-max* on line 2 are strictly increasing. Therefore, p_i will never choose t again. Since index is initially 0, $t(U) > 0$ for all UPDATE operations U. ◀

An UPDATE operation U is linearized at the first point that index has value at least $t(U)$. This occurs when some process, not necessarily the process performing U, performs index.*write-max*(t), for some $t \geq t(U)$. If multiple UPDATE operations are linearized at the same point, then they are linearized in increasing order of their indices. By Lemma 4, there will be no ties. An ISCAN operation is linearized when it performs index.*read-max*. A VIEW operation is linearized when it performs A$[t]$.*read*.

Since an UPDATE operation U begins with an index.*read-max* and $t(U)$ is chosen to be larger than the value it returns, index $< t(U)$ at the beginning of U's execution interval. Furthermore, since U ends with index.*write-max*$(t(U))$, index $\geq t(U)$ at the end of U's execution interval. Since the value of a max-register is non-decreasing, it follows that each UPDATE operation is linearized at a point within its execution interval.

▶ **Lemma 5.** *If an UPDATE operation U_1 is linearized before another UPDATE operation U_2, then $t(U_1) < t(U_2)$.*

Proof. Let X be the index.*write-max*(t) step at which U_1 is linearized, so that $t \geq t(U_1)$. If the index.*read-max* step of U_2 occurs after X, then the value returned by the index.*read-max* step of U_2 is at least t, so $t(U_2) > t \geq t(U_1)$. So, suppose that the index.*read-max* of U_2 occurs before X. If U_2 is also linearized at X, then, from the way the linearization order is defined when multiple UPDATE operations are linearized at the same step, $t(U_2) > t(U_1)$. Otherwise, U_2 is linearized after X so, by definition, $t(U_2) > t \geq t(U_1)$. ◀

▶ **Lemma 6.** *Let S be an ISCAN operation. If no UPDATE operation is linearized before S, then $t(S) = 0$. Otherwise, if U is the last UPDATE operation linearized before S, then $t(U) = t(S)$.*

Proof. Every UPDATE operation is linearized at no later than its index.*write-max* step. If S is linearized before every UPDATE operation, then its index.*read-max* step occurs before any index.*write-max* step and, hence, returns 0. So, suppose some UPDATE operation is linearized before S and let U be the last such UPDATE operation. Since S obtained index $t(S)$ from its index.*read-max* step, there was an UPDATE operation U' that previously performed index.*write-max*$(t(U'))$ with $t(U') = t(S)$. U' is linearized at no later than its index.*write-max* step. Hence, it is linearized before S. Suppose, for a contradiction, that $U \neq U'$, so that U is linearized between U' and S. By Lemma 5, $t(U) > t(U')$. By definition, there was a index.*write-max*(t) with $t \geq t(U)$ at the linearization point of U. This implies that $t(S) \geq t(U)$. This is a contradiction, since $t(U) > t(U') = t(S)$. ◀

▶ **Lemma 7.** *An UPDATE operation U is linearized before an ISCAN operation S if and only if $t(U) \leq t(S)$.*

Proof. Suppose an UPDATE operation U is linearized before an ISCAN S. Let U' be the last UPDATE operation linearized before S. By Lemma 6, $t(U') = t(S)$. If $U = U'$, then $t(U) = t(S)$. Otherwise, by Lemma 5, $t(U) < t(U') = t(S)$. Conversely, suppose that U is linearized after S. If no UPDATE operation is linearized before S, then $t(S) = 0$ and, by Lemma 4, $t(U) > t(S)$. So, suppose that some UPDATE operation is linearized before S and let U' be the last such UPDATE operation. Since U is linearized after U', by Lemma 5, $t(U) > t(U') = t(S)$. ◀

▶ **Lemma 8.** *For any two ISCAN operations S_1 and S_2, $t(S_1) < t(S_2)$ if and only if S_1 is linearized before S_2 and there is at least one UPDATE linearized between them.*

Proof. If $t(S_1) < t(S_2)$, then S_1 is linearized before S_2, since they are linearized at their respective index.*read-max* steps. By Lemma 4, there is an unique UPDATE operation U with $t(U) = t(S_2)$. Since $t(U) > t(S_1)$, Lemma 7 implies that U is linearized after S_1. Since U is the only UPDATE operation with $t(U) = t(S_2)$, the index.*write-max* step in U occurs before the index.*read-max* step in S_2. Since U is linearized at no later than its index.*write-max* step, it follows that U is linearized before S_2. Conversely, suppose S_1 is linearized before S_2 and there is at least one UPDATE operation U linearized between them. By Lemma 7, $t(S_1) < t(U) \leq t(S_2)$. ◀

▶ **Lemma 9.** *For any two ISCAN operations S_1 and S_2, $|t(S_1) - t(S_2)| \leq kn$, where k is the number of UPDATE operations linearized between them.*

Proof. Without loss of generality, assume S_1 is linearized before S_2. Let $U_1, U_2, \ldots, U_k$ be the UPDATE operations linearized between S_1 and S_2, in that order. By Lemma 7, $t(S_1) < t(U_1)$. By Lemma 6, $t(U_k) = t(S_2)$. By Lemma 5, $t(U_{i-1}) < t(U_i)$ for $2 \leq i \leq k$. Suppose the index.*read-max* step in U_1 returned a value $t > t(S_1)$. Then there was a index.*write-max*(t') after S_1 with $t(S_1) < t' < t(U_1)$, and the UPDATE operation U which performed this index.*write-max* would be linearized between S_1 and U_1, contradicting the definition of U_1. Therefore, the index.*read-max* step in U_1 returned a value which is at most $t(S_1)$. Similarly, for $2 \leq i \leq k$, the index.*read-max* step in U_i returned a value which is at most $t(U_{i-1})$. It follows by line 3 that $|t(U_1) - t(S_1)| \leq n$ and $|t(U_i) - t(U_{i-1})| \leq n$, for $2 \leq i \leq k$. It follows that $|t(S_1) - t(S_2)| = |t(S_1) - t(U_1) + t(U_1) - t(U_2) + \cdots + t(U_{k-1}) - t(U_k)| \leq nk$. ◀

▶ **Lemma 10.** *Let S be an ISCAN operation and let U be the last UPDATE operation linearized before S. Then every VIEW$(t(S), 1, 1, V)$ operation starting after S will set $V[1]$ to the value written by U.*

Proof. By Lemma 6, $t(U) = t(S)$. By Lemma 4, U is the unique UPDATE operation with $t(U) = t(S)$. After U completes, $\mathsf{A}[t(S)]$ contains the value written by U, since each UPDATE operation U' only writes to $\mathsf{A}[t(U')]$. The index.*read-max* step in S occurs after the index.*write-max* step in U, so U completes before S. It follows that any VIEW$(t(S), 1, 1, V)$ operation starting after S will set $V[1]$ to the value of $\mathsf{A}[t(S)]$, which contains the value written by U. ◀

▶ **Theorem 11.** *There is an implementation of a b-limited-use 1-component implicit snapshot object from w-bit registers and 1-bit registers, where w is the number of bits needed to represent each component of the snapshot. In the implementation, UPDATE consists of a write to a w-bit register and $\Theta(\log(bn+1))$ writes and reads on 1-bit registers, SCAN consists of $\Theta(\log(bn+1))$ reads on 1-bit registers, and VIEW consists of a single read on a w-bit register.*

Proof. Lemma 10 shows that a VIEW$(t, 1, 1, V)$ operation on an index t returned by an ISCAN operation S will set $V[1]$ to the value of the last UPDATE linearized before S. It follows that the implementation is linearizable. Lemma 8 shows that, for any two ISCANs S_1 and S_2, $t(S_1) < t(S_2)$ if and only if S_1 is linearized before S_2 and there is an UPDATE linearized between them, so the indices returned by ISCAN operations are correct. Lemma 9 shows that index is bounded by bn, since there can be at most b UPDATE operations between any two ISCANs. Hence a $(bn+1)$-bounded max register suffices and A needs at most $bn+1$ entries. By Theorem 1, a READ-MAX or WRITE-MAX on a $(bn+1)$-bounded max register requires $\Theta(\log(bn+1))$ reads and writes on 1-bit registers. ◀

4.2 An m-component implicit snapshot implementation

For $m > 1$, we obtain a b-limited-use m-component implicit snapshot recursively. Our implementation is essentially a modification of the implementation in Aspnes, Attiya, Censor-Hillel, and Ellen [4]. The result is a simpler implementation which uses substantially smaller registers.

Let snap_1 and snap_2 be b-limited-use c_1-component and c_2-component implicit snapshots, respectively. Let $\mathsf{indices}$ be a bounded 2-component max array. We describe a simple, but incorrect implementation of a b-limited-use $(c_1 + c_2)$-component implicit snapshot from snap_1 and snap_2 and then show how to fix it.

UPDATEs on the first c_1 components are handled by $\mathsf{snap}_1.\mathit{update}$ while UPDATEs on the last c_2 components are handled by $\mathsf{snap}_2.\mathit{update}$. The idea is that the ISCAN will use the 2-component max array to keep track of the most recent indices for snap_1 and snap_2. ISCAN performs $\mathsf{snap}_j.\mathit{iscan}$ to obtain index u_j and performs $\mathsf{indices}.\mathit{max\text{-}update}(j, u_j)$, for $j \in \{1, 2\}$. Then it performs $\mathsf{indices}.\mathit{max\text{-}scan}$ to obtain new indices (t_1, t_2), which it writes to the register $\mathsf{T}[t_1 + t_2]$, before finally returning $t_1 + t_2$. The VIEW operation for an index t returned by a previously completed ISCAN operation reads $\mathsf{T}[t]$ to obtain indices (t_1, t_2), and then calls $\mathsf{snap}_j.\mathit{view}$ on index t_j, for $j \in \{1, 2\}$, as necessary to get the appropriate components. By definition of a 2-component max-array, the indices (t_1, t_2) and (t'_1, t'_2) seen by two ISCAN operations as a result of their $\mathsf{indices}.\mathit{max\text{-}scan}$ steps are comparable component-wise. It follows that if $t_1 + t_2 = t'_1 + t'_2$, then $t_1 = t'_1$ and $t_2 = t'_2$. Hence, if two ISCAN operations write to $\mathsf{T}[t]$, for some index t, then they write the same value.

An UPDATE operation consists of a single $\mathsf{snap}_j.\mathit{update}$ step, at which it must be linearized. It is tempting to linearize an ISCAN seeing indices (t_1, t_2) from its $\mathsf{indices}.\mathit{max\text{-}scan}$ step at the first point in its execution interval that component 1 of $\mathsf{indices}$ has index t_1 and component 2 of $\mathsf{indices}$ has index t_2. It is possible to show that, with these linearization points, for any two ISCAN operation S_1 and S_2, if $t(S_1) < t(S_2)$, then S_1 is linearized before S_2 and there was an UPDATE operation linearized between them. The converse, however, does not hold. The problem is that, for S_2, some UPDATE operations to the first c_1 components may linearized between the first time that component 1 of $\mathsf{indices}$ has index t_1 and the first time that component 2 of $\mathsf{indices}$ has index t_2, and S_2 will fail to see these UPDATEs, even though it is linearized after them.

To fix this, we ensure that an UPDATE completes only after it knows it will be seen by all ISCANs linearized after it. Thus, after the UPDATE operation updates snap_j, it performs $\mathsf{snap}_j.\mathit{iscan}$ to obtain an index t, and then performs $\mathsf{indices}.\mathit{max\text{-}update}(j, t)$. Since UPDATEs now perform $\mathsf{snap}_j.\mathit{iscan}$ and $\mathsf{indices}.\mathit{max\text{-}update}(j, -)$, we can, in fact, remove the $\mathsf{snap}_j.\mathit{iscan}$ and $\mathsf{indices}.\mathit{max\text{-}update}(j, -)$ steps from ISCAN, for $j \in \{1, 2\}$. The pseudocode is presented in Algorithm 2.

Recall that, from Theorem 2, a bounded 2-component max array can be implemented from 1-bit registers. Thus, we will assume that all operations on $\mathsf{indices}$ are atomic and treat them as steps in the executions we consider.

An ISCAN operation S is linearized at its $\mathsf{indices}.\mathit{max\text{-}scan}$ step. We use $(t_1(S), t_2(S))$ to denote the value returned by this step, so that $t(S) = t_1(S) + t_2(S)$. An UPDATE operation which updates snap_j is linearized at the first point in its execution interval that component j of $\mathsf{indices}$ is at least the value returned in its $\mathsf{snap}_j.\mathit{iscan}$ step. More formally, for each UPDATE operation U, let $X(U)$ be the $\mathsf{snap}_j.\mathit{update}$ step in U, let $\mathcal{Y}(U)$ be the set of all $\mathsf{snap}_j.\mathit{iscan}$ steps occurring after $X(U)$, and let $\mathcal{Z}(U)$ be the set of all $\mathsf{indices}.\mathit{max\text{-}update}(j, t)$ steps, where $t = t(Y)$ for some $Y \in \mathcal{Y}(U)$. Let $Z(U)$ be the earliest step in $\mathcal{Z}(U)$. $Z(U)$ is part of some UPDATE operation U' which updates snap_j. Let $Y(U)$ be the $\mathsf{snap}_j.\mathit{iscan}$

Algorithm 2 A b-limited use $(c_1 + c_2)$-component implicit snapshot object.

```
procedure UPDATE($c, v$)
    $j \leftarrow$ if $c \leq c_1$ then 1 else 2
    $c \leftarrow$ if $c \leq c_1$ then $c$ else $c - c_1$
    $\mathsf{snap}_j.\mathit{update}(c, v)$
    $t \leftarrow \mathsf{snap}_j.\mathit{iscan}()$
    $\mathsf{indices}.\mathit{max\text{-}update}(j, t)$
procedure ISCAN( )
    $(t_1, t_2) \leftarrow \mathsf{indices}.\mathit{scan\text{-}max}()$
    $\mathsf{T}[t_1 + t_2].\mathit{write}((t_1, t_2))$
    return $t_1 + t_2$
procedure VIEW($t, i, j, V[1..j - i + 1]$)
    $(t_1, t_2) \leftarrow \mathsf{T}[t].\mathit{read}()$
    if $i \leq c_1$ then
        $\mathsf{snap}_1.\mathit{view}(t_1, i, \min\{j, c_1\}, V[1..\min\{c_1 - i + 1, j - i + 1\}])$
    if $j \geq c_1 + 1$ then
        $\mathsf{snap}_2.\mathit{view}(t_2, \max\{i - c_1, 1\}, j - c_1, V[\max\{1, c_1 - i + 2\}..j - i + 1])$
```

step in U'. Since there is a $Y \in \mathcal{Y}(U)$ with $t(Y) = t(Y(U))$, $X(U)$ cannot occur between Y and $Y(U)$, so $X(U) \rightarrow Y(U)$ and $Y(U) \in \mathcal{Y}(U)$. We linearize U at $Z(U)$. If multiple UPDATE operations are linearized at the same point, then linearize them in the order of their $\mathsf{snap}_j.\mathit{update}$ steps. A VIEW operation is linearized when it returns.

▶ **Lemma 12.** *An UPDATE operation U that updates snap_j is linearized before an ISCAN operation S if and only if $X(U)$ occurs before the first $\mathsf{snap}_j.\mathit{iscan}$ step returning $t_j(S)$.*

Proof. Suppose U is linearized before S. Let Y be the first $\mathsf{snap}_j.\mathit{iscan}$ step returning $t_j(S)$. Since U is linearized before S, its linearization point, $Z(U)$, occurs before the $\mathsf{indices}.\mathit{max\text{-}scan}$ step in S, so $t(Y(U)) \leq t_j(S) = t(Y)$. If $t(Y(U)) < t(Y)$, then $X(U) \rightarrow Y(U) \rightarrow Y$. Otherwise, if $t(Y(U)) = t(Y)$, then $Y \rightarrow Y(U)$ by definition of Y. Since $t(Y) = t(Y(U))$, there are no $\mathsf{snap}_j.\mathit{update}$ steps between Y and $Y(U)$, so we must have $X(U) \rightarrow Y \rightarrow Y(U)$.

Conversely, suppose $X(U)$ occurs before the first $\mathsf{snap}_j.\mathit{iscan}$ step Y returning $t_j(S)$. Let Z be the first $\mathsf{snap}_j.\mathit{max\text{-}update}(j, t_j(S))$ step, which is part of some UPDATE operation U', and let Y' be the $\mathsf{snap}_j.\mathit{iscan}$ step in U' which returned $t_j(S)$. By assumption, $X(U) \rightarrow Y \rightarrow Y'$, so $Y' \in \mathcal{Y}(U)$ and $Z \in \mathcal{Z}(U)$. Thus, U is linearized no later than Z, which is before the $\mathsf{indices}.\mathit{max\text{-}scan}$ step in S, so U is linearized before S. ◀

▶ **Lemma 13.** *Let S be an ISCAN operation, and let $U_1, \ldots, U_p$ be the UPDATE operations linearized before S that update snap_j, in the order in which they are linearized. Then $X(U_1) \rightarrow X(U_2) \rightarrow \cdots \rightarrow X(U_p)$ and $t(Y(U_1)) \leq t(Y(U_2)) \leq \cdots \leq t(Y(U_p)) = t_j(S)$.*

Proof. Let $1 \leq i < j \leq p$. Suppose, for a contradiction, that $t(Y(U_i)) > t(Y(U_j))$, so $X(U_j) \rightarrow Y(U_j) \rightarrow Y(U_i)$, $Y(U_i) \in \mathcal{Y}(U_j)$, and $Z(U_i) \in \mathcal{Z}(U_j)$. It follows that $Z(U_i) = Z(U_j)$, for otherwise U_j is linearized before U_i. Thus, $Y(U_i) = Y(U_j)$, so $t(Y(U_i)) = t(Y(U_j))$, which is a contradiction. To see that $t(Y(U_p)) = t_j(S)$, note that the UPDATE operation which performs the first $\mathsf{indices}.\mathit{max\text{-}update}(j, t_j(S))$ step Z, is linearized no later than Z, which is linearized before S.

Similarly, suppose for a contradiction that $X(U_j) \rightarrow X(U_i)$. It follows that $X(U_j) \rightarrow Y(U_i)$, $Y(U_i) \in \mathcal{Y}(U_j)$, and $Z(U_i) \in \mathcal{Z}(U_j)$. Thus, U_j would be linearized at no later than $Z(U_i)$, and it would be linearized before U_i since $X(U_j) \rightarrow X(U_i)$, a contradiction. ◀

▶ **Lemma 14.** *Consider any ISCAN operation S and any VIEW$(t(S), 1, c_1+c_2, V[1..c_1+c_2])$ operation. Then, after this VIEW operation completes, $V[c]$ is set to the value of the last UPDATE operation on component c linearized before S, or $\bot$ if no such operation exists, for all components c.*

Proof. We only consider when $c \in \{1, \ldots, c_1\}$; the case for $c \in \{c_1+1, \ldots, c_1+c_2\}$ is similar. Suppose that no UPDATE to component c is linearized before S. By Lemma 12, no $\mathsf{snap}_1.\mathit{update}(c, u)$ step occurs before the first $\mathsf{snap}_1.\mathit{iscan}$ step returning index $t_1(S)$. Thus, the view of snap_1 at index $t_1(S)$ has component c set to $\bot$, so $V[c] = \bot$.

Now, suppose some UPDATE to component c is linearized before S. Let U be the last such UPDATE, and let Y be the first $\mathsf{snap}_1.\mathit{iscan}$ step returning index $t_1(S)$. By Lemma 12, any UPDATE operation U' on component c such that $X(U') \to Y$ is linearized before S. Since U is the last UPDATE on component c linearized before S, Lemma 13 implies that $X(U') \to X(U)$. It follows that the view of snap_1 at index $t_1(S)$ has component c set to the value of $X(U)$, which is the value of U, so $V[c]$ is set to the value of U. ◀

▶ **Lemma 15.** *For any two ISCAN operations S_1 and S_2, $t(S_1) < t(S_2)$ if and only if S_1 is linearized before S_2 and there is at least one UPDATE linearized between them.*

Proof. If $t(S_1) < t(S_2)$, then S_1 is linearized before S_2 since they are linearized at their respective $\mathsf{indices}.\mathit{max\text{-}scan}$ steps. Furthermore, we must have either $t_1(S_1) < t_1(S_2)$ or $t_2(S_1) < t_2(S_2)$. Suppose $t_1(S_1) < t_1(S_2)$; the other case is similar. Since $t_1(S_1) < t_1(S_2)$, there was a $\mathsf{indices}.\mathit{max\text{-}update}(1, t_1(S_2))$ step Z before the start of S_2. Let U be the UPDATE operation which performed Z. Let Y be the first $\mathsf{snap}_1.\mathit{iscan}$ with $t(Y) = t_1(S_2)$. Since $t(Y) = t(Y(U))$, $X(U) \to Y$ and Lemma 12 implies that U is linearized before S. Since $t(Y(U)) = t_1(S_2) > t_1(S_1)$, Lemma 13 implies that U is linearized after S_1.

Conversely, suppose S_1 is linearized before S_2 and there is an UPDATE operation U linearized between them. Suppose that U updates snap_j. Since S_1 is linearized before S_2, $t(S_1) \leq t(S_2)$. Since U is linearized after S_1, by Lemma 12, $X(U)$ occurred after the first $\mathsf{snap}_j.\mathit{iscan}$ step Y returning $t_j(S_1)$. Since $Y \to X(U) \to Y(U)$, it follows that $t(Y(U)) > t(Y) = t_j(S_1)$. Furthermore, since U is linearized before S_2, by Lemma 13, $t(Y(U)) \leq t_j(S_2)$. Therefore $t(S_1) = t_j(S_1) + t_{3-j}(S_1) < t_j(S_2) + t_{3-j}(S_1) \leq t_j(S_2) + t_{3-j}(S_2) = t(S_2)$, where $t_2(S_1) \leq t_2(S_2)$ follows since the $\mathsf{indices}.\mathit{max\text{-}scan}$ step in S_1 occurs before the $\mathsf{indices}.\mathit{max\text{-}scan}$ step in S_2. ◀

▶ **Lemma 16.** *For every $\ell \geq 0$, there is an implementation of a b-limited use 2^ℓ-component implicit snapshot from w-bit registers, $\Theta(\log(bn+1))$-bit registers, and 1-bit registers, where w is the number of bits needed to represent each component of the snapshot. The implementation satisfies the following properties:*

1. *The ISCAN operation consists of $O((\log(bn+1))^2)$ reads and writes on 1-bit registers and a write to a $\Theta(\log(bn+1))$-bit register. Furthermore, for any two ISCAN operations S_1 and S_2, $|t(S_1) - t(S_2)| \leq kn$, where k is the number of UPDATE operations linearized between S_1 and S_2.*
2. *The UPDATE operation consists of $O((\log(bn+1))^2(\ell+1))$ reads and writes on 1-bit registers, ℓ writes to $\Theta(\log(bn+1))$-bit registers, and a write to a w-bit register.*
3. *The VIEW operation on an index returned by an ISCAN operation and a range (i, j) of components consists of $T_\ell(i,j) = 1 + \ell + 2(j-i) - \sum_{d=0}^{\ell-1}(y_d - x_d) \leq 1 + 2(\ell + j - i)$ reads on $\Theta(\log(bn+1))$-bit registers and $j - i + 1$ reads on w-bit registers, where $x_{\ell-1} \cdots x_0$ and $y_{\ell-1} \cdots y_0$ are the binary representations of $i-1$ and $j-1$, respectively.*

Proof. By induction on ℓ. The base case, when $\ell = 0$, holds by Theorem 11. Suppose now that the claim holds for ℓ and consider $\ell + 1$. We consider what happens when we build a b-limited use $2^{\ell+1}$-component implicit snapshot object from two b-limited use 2^ℓ-component implicit snapshot objects using Algorithm 2.

(1) Let S_1 and S_2 be any two ISCAN operations. For $i, j \in \{1, 2\}$, let Y_j^i be the first $\mathsf{snap}_j.\mathit{iscan}$ step returning $t_j(S_i)$. By assumption, $|t_j(Y_j^1) - t_j(Y_j^2)| \leq q_j n$, where k_j is the number of UPDATE operations which had their $\mathsf{snap}_j.\mathit{update}$ step linearized between Y_j^1 and Y_j^2. By Lemma 12, an UPDATE operation U which updates snap_j is linearized between S_1 and S_2 if and only if $X(U)$ occurs between Y_j^1 and Y_j^2. It follows that $k_1 + k_2 = k$. Thus, $|t(S_1) - t(S_2)| = |t_1(Y_1^1) - t_1(Y_1^2) + t_2(Y_2^1) - t_2(Y_2^2)| \leq |t(Y_1^1) - t(Y_1^2)| + |t(Y_2^1) - t(Y_2^2)| \leq (k_1 + k_2)n = kn$. Since $k \leq b$, this shows that $\mathsf{indices}$ can be $(bn + 1, bn + 1)$-bounded and T can be an array of $bn + 1$ $\Theta(\log(bn + 1))$-bit registers. By Theorem 2, $\mathsf{indices}$ can be implemented from 1-bit registers with step complexity $\Theta((\log(bn + 1))^2)$ for MAX-UPDATE and MAX-SCAN. Therefore, an ISCAN consists of $O((\log(bn + 1))^2)$ reads and writes on 1-bit registers, and a write to a $\Theta(\log(bn + 1))$-bit register.

(2) $\mathsf{snap}_j.\mathit{update}$ consists of $O((\log(bn + 1))^2(\ell + 1))$ reads and writes on 1-bit registers, ℓ writes to $\Theta(\log(bn + 1))$-bit registers, and a write to a w-bit register. $\mathsf{snap}_j.\mathit{iscan}$ consists of $O((\log(bn + 1))^2)$ reads and writes on 1-bit registers and a write to a $\Theta(\log(bn + 1))$-bit register. $\mathsf{indices}.\mathit{max\text{-}update}$ consists of $O((\log(bn + 1))^2)$ reads and writes on 1-bit registers. Thus, in total, an UPDATE consists of $O((\log(bn + 1))^2(\ell + 2))$ reads and writes on 1-bit registers, $\ell + 1$ writes to $\Theta(\log(bn + 1))$-bit registers, and a write to a w-bit register.

(3) By Algorithm 2, a VIEW operation at an index returned by an ISCAN operation on a range (i, j) of components consists of

$$T_{\ell+1}(i,j) = \begin{cases} T_\ell(i - 2^\ell, j - 2^\ell) + 1 & i > 2^\ell \\ T_\ell(i,j) + 1 & j \leq 2^\ell \\ T_\ell(i, 2^\ell) + T_\ell(1, j - 2^\ell) + 1 & \text{otherwise} \end{cases}$$

reads on $\Theta(\log(bn+1))$-bit registers. Let $x_\ell \cdots x_0$ and $y_\ell \cdots y_0$ be the binary representations of $i - 1$ and $j - 1$, respectively. If $i > 2^\ell$, then $x_\ell = y_\ell = 1$ and $0x_{\ell-1} \cdots x_0$ and $0y_{\ell-1} \cdots y_0$ are the binary representations of $(i - 2^\ell) - 1$ and $(j - 2^\ell) - 1$. In this case, it follows that

$$\begin{aligned} T_{\ell+1}(i,j) = T_\ell(i - 2^\ell, j - 2^\ell) + 1 &= 1 + \ell + 2((j - 2^\ell) - (i - 2^\ell)) - \sum_{d=0}^{\ell-1}(x_d - y_d) + 1 \\ &= 1 + (\ell + 1) + 2(j - i) - \sum_{d=0}^{\ell}(x_d - y_d)\,. \end{aligned}$$

If $j \leq 2^\ell$, then $x_\ell = y_\ell = 0$ and $0x_{k-1} \ldots x_0$ and $0y_{k-1} \cdots y_0$ are the binary representations of $i - 1$ and $j - 1$, respectively. In this case, it follows that

$$\begin{aligned} T_{\ell+1}(i,j) = T_\ell(i,j) + 1 &= 1 + \ell + 2(j - i) - \sum_{d=0}^{\ell-1}(y_d - x_d) + 1 \\ &= 1 + (\ell + 1) + 2(j - i) + \sum_{d=0}^{\ell}(y_d - x_d)\,. \end{aligned}$$

Otherwise, if $i \leq 2^\ell$ and $j > 2^\ell$, then $x_\ell = 0$ and $y_\ell = 1$ and $0x_{\ell-1}\ldots x_0$ and $0y_{\ell-1}\ldots y_0$ are the binary representations of $i-1$ and $(j-2^\ell)-1$, respectively. Note that the binary representation of $2^\ell - 1$ is $01\cdots1$ (one zero followed by $\ell-1$ ones). In this case, it follows that

$$\begin{aligned}
T_{\ell+1}(i,j) &= T_\ell(i,2^\ell) + T_\ell(1, j-2^\ell) + 1 \\
&= \left[1 + \ell + 2(2^\ell - i) - \sum_{d=0}^{\ell-1}(1-x_d)\right] + \left[1 + \ell + 2(j - 2^\ell - 1) - \sum_{d=0}^{\ell-1} y_d\right] + 1 \\
&= 1 + (\ell+1) + 2(j-i) - \sum_{d=0}^{\ell}(y_d - x_d)\,.
\end{aligned}$$

In all 3 cases, there are $j-i+1$ reads on w-bit registers. ◀

Inductively constructing implicit snapshots from implicit snapshots on a smaller number of components via Lemma 16, we obtain the following:

▶ **Theorem 17.** *There is an implementation of a b-limited-use m-component implicit snapshot from w-bit registers, $\Theta(\log(bn+1))$-bit registers, and 1-bit registers. The implementation satisfies the following:*

1. *The ISCAN operation consists of $O((\log(bn+1))^2)$ reads and writes on 1-bit registers and a write to a $\Theta(\log(bn+1))$-bit register.*
2. *The UPDATE operation consists of $O((\log(bn+1))^2 \log m)$ reads and writes on 1-bit registers, $\log m$ writes to $\Theta(\log(bn+1))$-bit registers, and a write to a w-bit register.*
3. *The VIEW operation on an index returned by an ISCAN operation and a range (i,j) of components consists of at most $1 + 2(\log m + j - i)$ reads on $\Theta(\log(bn+1))$-bit registers and $j-i+1$ reads on w-bit registers.*

Algorithm 3 A faster b-limited-use m-component implicit snapshot object.

```
procedure UPDATE($c, v$)
    old.$update(c, v)$
    index.$write\text{-}max$(old.$iscan()$)
procedure ISCAN( )
    return index.$read\text{-}max()$
procedure VIEW($t, i, j, V$)
    old.$view(t, i, j, V)$
```

We can reduce the step complexity of ISCAN to $O(\log(bn+1))$ reads on $\Theta(\log(bn+1))$-bit registers. The idea is to use the old implicit snapshot implementation from Theorem 17 and use a $(bn+1)$-bounded max register index, as in Algorithm 1, to store the most recent index. An UPDATE operation performs old.*update* to handle the actual update and then performs index.*write-max*(old.*iscan*()) to store the most recent index. An ISCAN simply returns the result of index.*read-max*. A VIEW operation calls old.*view*. See Algorithm 3 for the pseudocode.

Since old is linearizable, we can assume that the embedded old.*update* and old.*iscan* operations are atomic. UPDATEs which write t to index are linearized at the first point that index has value at least t. Ties are broken in order of the embedded old.*update* operations. ISCANs are linearized when they perform index.*read-max*.

Finally, by using the large register simulation of Aghazadeh, Golab, and Woelfel [2], we can obtain an implementation from k-bit registers, for $k \in \Omega(\log n)$. Putting this all together, we obtain the following:

▶ **Theorem 18.** *For $b \in n^{O(1)}$ and $k \in \Omega(\log n)$, there is an implementation of a b-limited-use m-component snapshot from k-bit registers with step complexity $O(w + (\log n)^2 \log m)$ for UPDATE and step complexity $O(mw + \log n)$ for SCAN, if each component of the snapshot consists of $\Theta(wk)$ bits.*

5 Conclusions

Aghazadeh, Golab, and Woelfel's [2] large register simulation requires registers with $\Omega(\log n)$ bits. Peterson's large register simulation works for arbitrarily small registers, but it is a single-writer register simulation and it is unclear if the simulation can be modified to efficiently implement interruptible reads. It would be interesting to find a large register simulation, which works for arbitrarily small registers, that can implement efficient interruptible reads.

It is not possible to directly apply our techniques to other, more efficient, snapshots [7, 6, 9] without significantly increasing their step complexities because they read too many large registers and they need most of the bits they read. We would like to investigate the possibility of modifying these snapshots to obtain a faster snapshot from $\Theta(\log n)$-bit registers.

For $b \in n^{O(1)}$, we showed how to implement a b-limited-use m-component snapshot from $\Theta(\log n)$-bit registers with UPDATE step complexity $\Theta(w + (\log n)^2 \log m)$ and SCAN step complexity $\Theta(mw + \log n)$, if each component of the snapshot consists of $\Theta(w \log n)$ bits. We had to use multi-writer registers. It would be interesting to see if it is possible to obtain similar results with only single-writer registers.

Acknowledgments. We thank Hagit Attiya and David Solymosi for helpful discussions. This research was supported by the Natural Science and Engineering Research Council of Canada.

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 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the Thirty-Third ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014.

3 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, 2012.

4 James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *Journal of the ACM*, 62(1):3:1–3:22, 2015.

5 Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the Twentieth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 336–343, 2008.

6 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

7 Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 29–40, 1993.

8 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

9 Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *Proceedings of the Eighth International Workshop on Distributed Algorithms (WDAG)*, pages 130–140, 1994.

10 Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

Anonymous Obstruction-Free (n, k)-Set Agreement with $n - k + 1$ Atomic Read/Write Registers[*]

Zohir Bouzid[1], Michel Raynal[2], and Pierre Sutra[3]

1 IRISA, Université de Rennes, Rennes, France
2 IRISA, Université de Rennes, Rennes, France; and
Institut Universitaire de France, Paris, France
3 University of Neuchâtel, Neuchatel, Switzerland; and
Télécom SudParis, CNRS, Université Paris-Saclay, Paris, France

Abstract

The k-set agreement problem is a generalization of the consensus problem. Namely, assuming that each process proposes a value, every non-faulty process should decide one of the proposed values, and no more than k different values should be decided. This is a hard problem in the sense that we cannot solve it in an asynchronous system, as soon as k or more processes may crash. One way to sidestep this impossibility result consists in weakening the termination property, requiring that a process must decide a value only if it executes alone during a long enough period of time. This is the well-known obstruction-freedom progress condition.

Consider a system of n *anonymous asynchronous* processes that communicate through atomic *read/write registers*, and such that *any number of them may crash*. In this paper, we address and solve the challenging open problem of designing an obstruction-free k-set agreement algorithm using only $(n-k+1)$ atomic registers. From a shared memory cost point of view, our algorithm is the best algorithm known so far, thereby establishing a new upper bound on the number of registers needed to solve the problem, and in comparison to the previous upper bound, its gain is $(n-k)$ registers. We then extend this algorithm into a space-optimal solution for the repeated version of k-set agreement, and an x-obstruction-free solution that employs $(n - k + x)$ atomic registers (with $1 \leq x \leq k < n$).

1998 ACM Subject Classification D.1.3 [Software] Concurrent Programming, F.2 [Theory of Computation] Analysis of Algorithms and Problem Complexity, F.2.2 [Theory of Computation] Nonnumerical Algorithms and Problems

Keywords and phrases Anonymous processes, Asynchronous system, Atomic read/write register, Consensus, Fault-tolerance, k-Set agreement, Obstruction-freedom, Upper bound

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.18

1 Introduction

Due to failures, concurrent processes have to deal not only with finite asynchrony, i.e., finite but arbitrary process speed, but also with infinite asynchrony. In this context, mutex-based

[*] This work has been partially supported by the French ANR project DISPLEXITY devoted to computability and complexity in distributed computing, the Franco-German ANR project DISCMAT devoted to connections between mathematics and distributed computing, as well as the European Commission's Seventh Framework Program (FP7) under grant agreement No. 318809 (LEADS).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 18; pp. 18:1–18:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

synchronization becomes useless, and pioneering works in *fault-tolerant* distributed computing, e.g., [21, 25], have instead promoted the design of concurrent algorithms [19, 26, 29].

A first challenge: multi-writer registers. When processes communicate with *Single-Writer Multi-Reader* (SWMR) atomic registers, a concurrent algorithm usually associates each process with a register. In the case where processes communicate with *Multi-Writer Multi-Reader* (MWMR) atomic registers, as any process can write any register, the previous association is no longer granted for free. To still benefit from existing SWMR registers-based solutions, a classical reduction consists in emulating SWMR registers on top of MWMR registers. In a system of n processes, it is shown [7, 9] that $(2n-1)$ MWMR atomic registers are needed to wait-free [16] simulate one SWMR atomic register, and that n MWMR atomic registers are needed if the simulation is required to be only non-blocking [20].[1] As a consequence, the simulation approach becomes irrelevant if the system provides less than n MWMR registers. In this context, the present paper focuses on what we name *genuine* concurrent algorithms, where "genuine" means "without simulating SWMR registers on top of MWMR registers". As underlined in [8], the design of genuine algorithms based on MWMR registers is still in its infancy, and sometimes resembles "black art" in the sense that the underlying intuition is difficult to grasp and formulate.

A second challenge: anonymity. Some algorithms based on MWMR registers, e.g., [26], require processes to write control values that include their identities. On the contrary, in an *anonymous* system, processes have no identity, the same code, and the same initialization of their local variables. Hence, they are in a strong sense identical. In such a context, the core question that interests us is the following: "Is it possible to solve a given problem with MWMR registers and anonymous processes, and if the answer is "yes", how many registers do we need ?"

Consensus and k-set agreement. We focus on the k-set agreement problem in a system of n processes. This problem introduced in [6], and denoted (n,k)-set agreement in the following, is a generalization of consensus, which corresponds to the case where $k=1$. Assuming that each participating process proposes a value, every non-faulty process must decide a value (termination), which was proposed by some process (validity), and at most k different values are decided (agreement).

Impossibility results and the case of obstruction-freedom. Designing a deterministic wait-free consensus in an asynchronous system prone to even a single crash failure is not possible [12, 23]. If now k or more processes may crash, there is no deterministic wait-free read/write solution to (n,k)-set agreement [4, 18, 27]. As we are interested in the computing power of *pure read/write* asynchronous systems, we neither want to enrich the underlying system with additional power (e.g., synchrony assumptions, random numbers, or failure detectors), nor impose constraints on the input vector collectively proposed by the processes. To sidestep the above impossibility result, we thus consider a progress property weaker than wait-freedom, namely *obstruction-freedom* [17]. For (n,k)-set agreement, this property states that a process decides a value only if it executes solo during a "long enough" period of time without

[1] "Wait-free" means that any read or write invocation on the SWMR register that is built must terminate if the invoking process does not crash [16]. "Non-blocking" means that at least one process that does not crash returns from all its read and write invocations [20].

interruption. The notion of x-obstruction-freedom [30] generalizes this idea to any group of at most x processes.

Contributions of the paper. This paper details a *genuine obstruction-free* algorithm solving the (n, k)-set agreement problem in an *asynchronous anonymous read/write* system where any number of processes may crash. Our algorithm makes use of $(n - k + 1)$ MWMR registers, i.e., exactly n registers for consensus. For (n, k)-set agreement, the best lower bound known so far [10] is $\Omega(\sqrt{\frac{n}{k} - 2})$, while the best obstruction-free (n, k)-set agreement algorithm requires $2(n - k) + 1$ MWMR registers [8, 10]. As a consequence, our algorithm provides a gain of $(n - k)$ MWMR registers. In the case of consensus, Gelashvili [14] proved recently that $n/20$ registers are necessary, and Zhu [31] improved this bound to $n - 1$. Hence, our algorithm is up to an additive factor of 1 close to the best known lower bound.

In the *repeated* version of the (n, k)-set agreement problem, processes participate in a sequence of (n, k)-set agreement instances. It was recently proved [10] that $(n - k + 1)$ atomic registers are necessary to solve repeated (n, k)-set agreement. This paper shows that a simple modification of our base construction solves *repeated* (n, k)-set agreement without additional atomic registers. The resulting algorithm is thus optimal, closing the gap on previous proposed upper bounds for this problem.

Our construction is round-based, following the pattern “snapshot; local computation; write”, where the snapshot and write operations occur on the $(n-k+1)$ MWMR registers. This pattern is reminiscent of the one named “look; compute; move” found in robot algorithms [28]. Interestingly, processes do not maintain any local information between successive rounds. In this sense, our algorithm is *locally memoryless*. Each register contains a quadruplet consisting of a round number, two control bits, and a proposed value. The algorithm exploits a partial order over the quadruplets. The way a process computes a new quadruplet is the key of our algorithm. The variation for repeated (n, k)-set agreement employs sixuplets.

Roadmap. Section 2 presents the computing model and definitions used in this paper. Section 3 depicts a base anonymous obstruction-free algorithm solving consensus; this algorithm captures the essence of our solution. We prove its correctness in Section 4. Section 5 extends our algorithm to solve (n, k)-set agreement, We address the case where (n, k)-set agreement is used repeatedly in Section 6. Section 7 considers the x-obstruction-freedom progress condition, and presents a solution using $(n - k + x)$ registers. We conclude in Section 8. Due to space constraints, we defer some details to our companion technical report [5].

2 Context & Problem Definition

2.1 Computing Model

We assume a distributed system of n asynchronous processes $\{p_1, \ldots, p_n\}$. When considering a process p_i, we name integer i its *index*. Indexes are used to ease the exposition from an external observer point of view. Processes do not have identities and execute the very same code. We assume that they know the value n.

Let $\mathbb{T}$ denote the increasing sequence of time instants (observable only from an external point of view). At each instant, a unique process is activated to execute a step. A *step* consists in a read or a write to a register (access to the shared memory) possibly followed by a finite number of internal operations (on local variables).

Up to $(n-1)$ processes may crash. A crash is an unexpected halting. After it has crashed (if it ever does), a process remains crashed forever. From a terminology point of view, and given an execution, a *faulty* process is a process that crashes, and a *correct* process is a process that does not crash.[2]

In addition to processes, the computing model includes a communication medium made up of m multi-writer/multi-reader (MWMR) registers.[3] Registers are encapsulated in an array denoted $REG[1..m]$. The registers are *atomic*. This means that read and write operations appear as if they have been executed sequentially, and this sequence (a) respects the real-time order of non-concurrent operations, and (b) is such that each read returns the value written by the closest preceding write operation [22]. When considering some concurrent object defined from a sequential specification, atomicity is named *linearizability* [20].

From atomic registers to a snapshot object. At the upper layer (where consensus and (n,k)-set agreement are solved), we use the array $REG[1..m]$ to construct a snapshot object [1]. This object, denoted REG hereafter, provides processes with the operations $\mathsf{write}()$ and $\mathsf{snapshot}()$. When a process invokes $REG.\mathsf{write}(x,v)$, it deposits the value v in $REG[x]$. When it invokes $REG.\mathsf{snapshot}()$ it obtains the content of the whole array. The snapshot object is linearizable, i.e., every invocation of $REG.\mathsf{snapshot}()$ appears as instantaneous. For the REG object, a linearization is a sequence of write and snapshot operations.

An anonymous non-blocking (hence obstruction-free) implementation of a snapshot object is described in [15]. This implementation does not require additional atomic registers. In the following, we consider that the snapshot object REG is implemented using this algorithm.

2.2 Obstruction-free consensus and obstruction-free (n,k)-set agreement

Obstruction-free consensus. An obstruction-free consensus object is a one-shot object that provides each process with a single operation denoted $\mathsf{propose}()$. This operation takes a value as input parameter and returns a value.

"*One-shot*" means that a process invokes $\mathsf{propose}()$ at most once. When a process invokes $\mathsf{propose}(v)$, we say that it "proposes v". When the invocation of $\mathsf{propose}()$ returns value v', we say that the invoking process "decides v'". A process executes "solo" when it keeps on executing while the other processes have stopped their execution (at any point of their algorithm). The obstruction-free consensus problem is defined by the following properties (that is, to be correct, any obstruction-free algorithm must satisfy such properties).

- Validity. If a process decides a value v', this value was proposed by a process.
- Agreement. No two processes decide different values.
- OB-termination. If there is a time after which a process executes solo, it decides a value.
- SV-termination[4]. If a single value is proposed, all correct processes decide.

Validity relates outputs to inputs. Agreement relates the outputs. Termination states the conditions under which a correct process must decide. There are two cases. The first is

[2] No process knows if it is correct or faulty. This is because, before crashing, a faulty process behaves as a correct one.

[3] As pointed out in the introduction, we recall that anonymity prevents processes from using single-writer/multi-reader registers.

[4] This termination property, which relates termination to the input values, is not part of the classical definition of the obstruction-free consensus problem. It is an additional requirement which demands termination under specific circumstances that are independent of the concurrency pattern.

```
function sup($T$) is
(S1) let $\langle r, \ell evel, -, v\rangle$ be max($T$);
(S2) let $vals(T)$ be $\{w \mid \exists\langle r, -, -, w\rangle \in T\}$;
(S3) let $conf\ell ict1(T)$ be $\exists\ \langle r, -, \texttt{true}, -\rangle \in T$;
(S4) let $conf\ell ict2(T)$ be $|vals(T)| > 1$;
(S5) let $conf\ell ict(T)$ be $conf\ell ict1(T) \vee conf\ell ict2(T)$;
(S6) return$\big(\langle r, \ell evel, conf\ell ict(T), v\rangle\big)$.
```

Figure 1 The function sup().

related to obstruction-freedom. The second one is independent of the concurrency and failure pattern; it is related to the input value pattern.

Obstruction-free (n, k)-set agreement. An obstruction-free (n, k)-set agreement object is a one-shot object which has the same validity, OB-termination, and SV-termination properties as consensus, and for which we replace the agreement property with:

- Agreement. At most k different values are decided.

As for consensus, SV-termination property is a new property strengthening the classical definition of k-set agreement [6].

In what follows, we describe first an obstruction-free anonymous algorithm that solves the consensus problem, then we extend it to address (n, k)-set agreement.

3 Obstruction-free Anonymous Consensus Algorithm

Our consensus algorithm is detailed in Figure 2. As indicated in the Introduction, its essence is captured by the quadruplets that can be written in the MWMR atomic registers.

Shared memory. The shared memory is made up of a snapshot object REG, composed of $m = n$ MWMR atomic registers. Each of them contains a quadruplet initialized to $\langle 0, \texttt{down}, \texttt{false}, \perp\rangle$. The meaning of these fields is the following.

- The first field, denoted rd, is a round number.
- The second field, denoted $\ell v\ell$ (level), has a value in $\{\texttt{up}, \texttt{down}\}$, where $\texttt{up} > \texttt{down}$.
- The third field, denoted $cf\ell$ (conflict), is a Boolean (initially equals to $\texttt{false}$). We assume $\texttt{true} > \texttt{false}$.
- The last field, denoted $va\ell$, is initialized to $\perp$, and then contains always a proposed value. It is assumed that the set of proposed values is totally ordered, and that $\perp$ is smaller than any proposed value.

When considering the lexicographical ordering, it is easy to see that all the possible quadruplets $\langle rd, \ell v\ell, cf\ell, va\ell\rangle$ form a totally ordered set. This total order, and its reflexive closure, are denoted "$<$" and "$\leq$", respectively.

Notion of conflict and the function sup(). The function sup(), defined in Figure 1, plays a central role in our algorithm. It takes a non-empty set of quadruplets T as input parameter, and returns a quadruplet, which is the supremum of T, defined as follows.

Let $\langle r, \ell evel, -, v\rangle$ be the maximal element of T according to the lexicographical ordering (line S1), and $vals(T)$ be the values in the quadruplets of T associated with the maximal round number r (line S2). The set T is *conflicting* if one of the two following cases occurs (line S5).

```
operation propose($v_i$) is
(01)  repeat forever
(02)     $view \leftarrow REG$.snapshot();
(03)     case $(\exists r > 0, va\ell : \forall x : view[x] = \langle r, \mathtt{up}, \mathtt{false}, va\ell \rangle)$ then
                    return($va\ell$)
(04)           $(\exists r > 0, va\ell : \forall x : view[x] = \langle r, \mathtt{down}, \mathtt{false}, va\ell \rangle)$ then
                    $REG$.write$(1, \langle r+1, \mathtt{up}, \mathtt{false}, va\ell \rangle)$
(05)           $(\exists r > 0, va\ell, \ell evel : \forall x : view[x] = \langle r, \ell evel, \mathtt{true}, va\ell \rangle)$ then
                    $REG$.write$(1, \langle r+1, \mathtt{down}, \mathtt{false}, va\ell \rangle)$;
(06)           otherwise let $\langle r, \ell evel, cf\ell, va\ell \rangle$
                                 $\leftarrow$ sup$(view[1], \cdots, view[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle)$;
(07)                 $x \leftarrow$ smallest index such that $view[x] \neq \langle r, \ell evel, cf\ell, va\ell \rangle$;
(08)                 $REG$.write$(x, \langle r, \ell evel, cf\ell, va\ell \rangle)$
(09)     end case
(10) end repeat.
```

Figure 2 Anonymous obstruction-free Consensus.

- There is a quadruplet $X = \langle r, -, \mathtt{true}, - \rangle$ in T (line S3). In this case, there is a quadruplet $X \in T$ whose round number is the highest ($X.rd = r$), and whose conflict field $X.\ell v\ell = \mathtt{true}$. We then say that the conflict is "inherited".
- There are at least two quadruplets X and Y in T, that have the highest round number in T (i.e., $X.rd = Y.rd = r$), and that contain two different values (i.e., $X.va\ell \neq Y.va\ell$) (lines S2 and S4). In such a case, we say that the conflict is "discovered".

Function sup(T) first checks whether T is conflicting (lines S2–S5). Then, it returns at line S6 the quadruplet $\langle r, \ell evel, conf\ell ict(T), v \rangle$, where $conf\ell ict(T)$ indicates if the input set T is conflicting (line S5). Let us notice that, since $\mathtt{true} > \mathtt{false}$, the quadruplet returned by sup(T) is always greater than, or equal to, the greatest element in T, i.e., sup$(T) \geq$ max(T).

The algorithm. Our base construction is pretty simple, and consists in an appropriate management of the snapshot object REG, so that the n quadruplets it contains (a) never allow validity or agreement to be violated, and (b) eventually allow termination under good circumstances (which occur when obstruction-freedom is satisfied or when a single value is proposed).

In Figure 2, when a process p_i invokes proposes(v_i), it enters a loop that it will exit at line 03 (provided it terminates) with the statement return$(va\ell)$, where $va\ell$ is the decided value. After entering the loop, a process issues a snapshot and assigns the returned array to its local variable $view[1..n]$ (line 02). Then, there are two main cases according to the value stored in $view$.

- Case 1 (lines 03–05). All entries of $view_i$ contain the same quadruplet $\langle r, \ell evel, conflict, va\ell \rangle$, and $r > 0$. Then, there are three sub-cases to consider.
  - Case 1.1. If the level is $\mathtt{up}$ and the conflict is $\mathtt{false}$, the invoking process decides the value $va\ell$ (line 03).
  - Case 1.2. If the level is $\mathtt{down}$ and the conflict field is $\mathtt{false}$, process p_i enters the next round by writing $\langle r+1, \mathtt{up}, \mathtt{false}, val \rangle$ in the first entry of REG (line 04).
  - Case 1.3. If there is a conflict, p_i enters the next round by writing $\langle r+1, \mathtt{down}, \mathtt{false}, val \rangle$ in the first entry of REG (line 05).

- Case 2 (lines 06–08). Not all the entries of $view_i$ are equal, or one of them contains a tuple $\langle 0, -, -, - \rangle$. In such a case, p_i first calls $\mathtt{sup}(view[1], \cdots, view[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle)$ (line 06), which returns a quadruplet X greater than all the input quadruplets, or equal to the greatest of them. As we have seen previously, this quadruplet X may inherit or discover a conflict. Moreover, as $\langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle$ is an input parameter of $\mathtt{sup}()$, $X.val$ cannot equal $\perp$. As none of the predicates at lines 03–05 is satisfied, at least one entry of $view[1..n]$ is different than X. Process p_i writes then X into $REG[x]$, where, from its point of view, x is the first entry of REG whose content differs from X (lines 07–08).

The underlying operational intuition. To understand the intuition that underlies our algorithm, let us first consider the very simple case where a single process p_i executes the algorithm. From its first invocation of REG.snapshot() (line 02), it obtains a view $view$ in which all the elements are equal to $\langle 0, \mathtt{down}, \mathtt{false}, \perp \rangle$. Hence, p_i executes line 06, where the invocation of $\mathtt{sup}()$ returns the quadruplet $\langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle$, that is written into $REG[1]$ at line 08. Then, during the second round, p_i computes with the help of function $\mathtt{sup}()$ again the quadruplet $\langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle$, and p_i writes it into $REG[2]$; etc., until p_i writes $\langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle$ in all the atomic registers of $REG[1..n]$. When this occurs, p_i obtains at line 02 a view where all the elements equal $\langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle$. It consequently executes line 04 and writes $\langle 2, \mathtt{up}, \mathtt{false}, v_i \rangle$ in $REG[1]$. Then, during the following rounds, process p_i writes $\langle 2, \mathtt{up}, \mathtt{false}, v_i \rangle$ in the other registers of REG (line 08). When this is done, p_i obtains a snapshot containing solely $\langle 2, \mathtt{up}, \mathtt{false}, v_i \rangle$, and when this occurs, p_i executes line 03 where it decides the value v_i.

Let us now consider the case where, while p_i is executing, another process p_j invokes propose(v_j) with $v_j = v_i$. It is easy to see that, in such a case, p_i and p_j collaborate to fill in REG with the same quadruplet $\langle 2, \mathtt{up}, \mathtt{false}, v_i \rangle$. If $v_j \neq v_i$, depending on the concurrency pattern, a conflict may occur. For instance, it occurs if REG contains both $\langle 1, \mathtt{down}, \mathtt{false}, v_i \rangle$ and $\langle 1, \mathtt{down}, \mathtt{false}, v_j \rangle$. If a conflict appears, it will be propagated from round to round, until a process executes alone a higher round.

▶ Remark 1. We first notice that no process needs to memorize in its local memory the values that it will use during the next rounds. Not only processes are anonymous, but their code is also memoryless (no persistent variables). The snapshot object REG constitutes the whole memory of the system. Hence, as defined in the Introduction, the algorithm is locally memoryless. In this sense, and from a locality point of view, it has a "functional" flavor.

▶ Remark 2. Let us consider the n-bounded concurrency model [2, 24]. This model is made up of an arbitrary number of processes, but, at any time, there are at most n processes executing steps. This allows processes to leave the system and other processes to join it as long as the concurrency degree does not exceed n.

The previous algorithm works without modification in such a model. A proposed value is now a value proposed by any of the N processes that participate in the algorithm. Hence, if $N > n$, the number of proposed values can be greater than the upper bound n on the concurrency degree. This versatility dimension of our algorithm is a direct consequence of the previous "locally memoryless" property.

4 Proof of the Algorithm

In this section, we present the correctness proof of our obstruction-free anonymous consensus algorithm. After a few definitions provided in Section 4.1, Section 4.2 shows that a relation "$\sqsupseteq$" defined over the quadruplets is a partial order. This relation is central to prove the key properties of our algorithm. Such properties are established in Sections 4.3 and 4.4. Then, based on these properties, Section 4.5 shows that our algorithm is correct. Notice that, due to space restrictions, we state some lemmata without their corresponding proofs. The interested reader will find them in our companion technical report [5].

4.1 Definitions and notations

Let $\mathcal{E}$ be a set of quadruplets that can be written in *REG*. Given $X \in \mathcal{E}$, its four fields are denoted $X.rd, X.\ell v\ell, X.cf\ell$ and $X.va\ell$, respectively. Relations $>$ and $\geq$ refer to the lexicographical ordering over $\mathcal{E}$. Moreover, where appropriate, we consider the array $view[1..n]$ as the set $\{view[1], \cdots, view[n]\}$.

▶ **Definition 3.** Let $X, Y \in \mathcal{E}$.

$$X \sqsupset Y \overset{def}{=} (X > Y) \wedge [(X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Y.cf\ell \wedge X.va\ell = Y.va\ell)].$$

At the operational level the algorithm ensures that the quadruplets it generates are totally ordered by the relation $>$. Differently, the relation $\sqsupset$ (which is a partial order on these quadruplets, see Section 4.2) captures the relevant part of this total order, and is consequently the key cornerstone on which the proof of our algorithm relies.

When $X \sqsupset Y$ holds, we say that "X *strictly dominates* Y". Similarly, X *dominates* Y, written $X \sqsupseteq Y$, when $(X \sqsupset Y)$ or $(X = Y)$ holds. Relations $\sqsubset$ and $\sqsubseteq$ are defined in the natural way.

▶ **Definition 4.** Given a set of quadruplets T, we shall say that T is *homogeneous* when it contains a single element, say X. We then write it "T is $\mathcal{H}(X)$".

▶ Notation 1. The value, at time τ, of the local variable α of a process p_i is denoted α_i^τ. Similarly the value of an atomic register $REG[x]$ at time τ is denoted $REG^\tau[x]$, and the value of REG at time τ is denoted REG^τ.

▶ Notation 2. We note $\mathcal{W}(x, X)$ the writing of a quadruplet X in the register $REG[x]$.

▶ **Definition 5.** We say "a process p_j *covers* $REG[x]$ at time τ" when its next non-local step after time τ is $\mathcal{W}(x, X)$, where X is the quadruplet which is written. In this case we also say "$\mathcal{W}(x, X)$ *covers* $REG[x]$ at time τ" or "$REG[x]$ *is covered* by $\mathcal{W}(x, X)$ at time τ".

Let us notice that if p_j covers $REG[x]$ at time τ, then τ necessarily lies between the last snapshot issued by p_j at line 02 and its planned write $\mathcal{W}(x, X)$ that will occur at line 04, 05, or 08.

4.2 The relation $\sqsupseteq$ is a partial order

▶ **Lemma 6.** $((X \sqsupset Y \sqsupset Z) \wedge (X.rd = Y.rd = Z.rd)) \Rightarrow (X.cf\ell \vee (\neg Z.cf\ell \wedge X.va\ell = Z.va\ell))$.

▶ **Lemma 7.** $\sqsupseteq$ *is a partial order.*

4.3 Extracting the relations $\sqsupset$ and $\sqsupseteq$ from the algorithm

The definition of sup() appears in Figure 1.

▶ **Lemma 8.** *Let T be a set of quadruplets. For every $X \in T$: $\mathsf{sup}(T) \sqsupseteq X$.*

▶ **Lemma 9.** *If p_i executes $\mathcal{W}(-, Y)$ at time τ, then for every $X \in view_i^\tau$: $Y \sqsupseteq X$.*

▶ **Lemma 10.** *Let us assume that no process is covering $REG[x]$ at time τ. For every write $\mathcal{W}(-, X)$ that (a) occurs after τ and (b) was not covering a register of REG at time τ, we have $X \sqsupseteq REG^{\tau}[x]$.*

Proof. The proof is by contradiction. Let p_i be the first process that executes a write $\mathcal{W}(-, X)$ contradicting the lemma. This means that $\mathcal{W}(-, X)$ is not covering a register of REG at time τ and $X \not\sqsupseteq REG^{\tau}[x]$. Let this write occur at time $\tau_2 > \tau$. Thus, all writes that take place between τ and τ_2 comply with the lemma. We derive a contradiction by showing that $X \sqsupseteq REG^{\tau}[x]$.

Let $\tau_1 < \tau_2$ be the linearization time of the last snapshot taken by p_i (line 02) before executing $\mathcal{W}(-, X)$. Since $\mathcal{W}(-, X)$ was not covering a register of REG at time τ, the snapshot preceding this write was necessarily taken after τ. That is, $\tau_1 > \tau$, and we have $\tau_2 > \tau_1 > \tau$.

According to Lemma 9, $X \sqsupseteq view_i^{\tau_2}[x]$. But since the snapshot returning $view_i^{\tau_2}$ is linearized at τ_1, it follows that $view_i^{\tau_2} = REG^{\tau_1}$. Therefore, we have $X \sqsupseteq REG^{\tau_1}[x]$ (assertion R).

In the following we show that $REG^{\tau_1}[x] \sqsupseteq REG^{\tau}[x]$. If $REG[x]$ was not updated between τ and τ_1, then $REG^{\tau_1}[x] = REG^{\tau}[x]$ and the claim follows. Otherwise, if $REG[x]$ was updated between τ and τ_1, the content of $REG^{\tau_1}[x]$, let it be Y, is the result of a write $\mathcal{W}(x, Y)$ that occurred between τ and τ_1 and that was not covering a register of REG at time τ (remember that no write is covering $REG[x]$ at time τ). We assumed above that τ_2 is the first time at which the lemma is contradicted. Hence the write $\mathcal{W}(x, Y)$, which occurs before τ_2, complies with the requirements of the lemma. It follows that $Y \sqsupseteq REG^{\tau}[x]$, and we consequently have $REG^{\tau_1}[x] \sqsupseteq REG^{\tau}[x]$.

But it was shown above (see assertion R) that $X \sqsupseteq REG^{\tau_1}[x]$. Hence, due to the transitivity of the relation $\sqsupseteq$ (Lemma 7), we obtain $X \sqsupseteq REG^{\tau}[x]$, a contradiction that concludes the proof of the lemma. ◀

▶ **Lemma 11.** *Let τ and $\tau' \geq \tau$ be two time instants. If $REG^{\tau'}$ is $\mathcal{H}(Y)$, then there exists $X \in REG^{\tau}$ such that $Y \sqsupseteq X$.*

The following two lemmata are corollaries of Lemma 11.

▶ **Lemma 12.** *If REG^{τ} is $\mathcal{H}(X)$, $REG^{\tau'}$ is $\mathcal{H}(Y)$, and $\tau' \geq \tau$, then $Y \sqsupseteq X$.*

▶ **Lemma 13.** *If REG^{τ} is $H(X)$, $REG^{\tau'}$ is $\mathcal{H}(Y)$, $\tau' \geq \tau$, $(Y.rd = X.rd)$ and $(\neg Y.cf\ell)$ then $(Y.va\ell = X.va\ell)$.*

4.4 Exploiting homogeneous snapshots

▶ **Lemma 14.** $[(X \in REG^{\tau}) \wedge (X.\ell v\ell = \mathtt{up})] \Rightarrow \big(\exists\, \tau' < \tau$*: $REG^{\tau'}$ is $\mathcal{H}(Z)$, where $Z = \langle X.rd - 1, \mathtt{down}, \mathtt{false}, X.va\ell\rangle\big)$.*

Proof. Let us first show that there is a process that writes the quadruplet X' into REG, with $X' = \langle X.rd, X.\ell v\ell, \mathtt{false}, X.va\ell\rangle$. We have two cases depending on the value of $X.cf\ell$.

- If $X.cf\ell = \mathtt{false}$, then let $X' = X$. Since $X.\ell v\ell = X'.\ell v\ell = \mathtt{up}$, X was necessarily written into REG by some process (let us recall that the initial value of each register of REG is $\langle 0, \mathtt{down}, \mathtt{false}, \perp\rangle$).
- If $X.cf\ell = \mathtt{true}$, let us consider the time τ_1 at which X was written for the first time into REG, say by p_i. Since $X.\ell v\ell = \mathtt{up}$, both τ_1 and p_i are well defined. This write of X happens necessarily at line 08 (If it was at line 04 or 05, we would have $X.cf\ell = \mathtt{false}$).

Therefore, X was computed at line 06 by the function $\mathsf{sup}()$. Namely we have $X = \mathsf{sup}(T)$, where the set T is equal to $\{view^{\tau}[1], \cdots, view^{\tau}[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle\}$. Observe that $X \notin T$, otherwise X would not be written for the first time at τ_1. Let $X' = \mathsf{max}(T)$. Since $X \notin T$, it follows that $X \neq X'$. Due to line S6 of the function $\mathsf{sup}()$, X and X' differ only in their conflict field. Therefore, as $X.cf\ell = \mathtt{true}$, it follows that $X'.cf\ell = \mathtt{false}$. Finally, as $X'.\ell v\ell = \mathtt{up}$ and all registers of REG are initialized to $\langle 0, \mathtt{down}, \mathtt{false}, \perp\rangle$, it follows that X' was necessarily written into REG by some process.

In both cases, there exists a time at which a process writes $X' = \langle X.rd, X.\ell v\ell, \mathtt{false}, X.va\ell\rangle$ into REG. Let us consider the first process p_i that does so. This occurs at some time $\tau_2 < \tau$. As $X'.\ell v\ell = \mathtt{up}$, this write can occur only at line 04 or line 08.

We first show that this write occurs necessarily at line 04. Assume for contradiction that the write of X' into REG happens at line 08. In this case, the quadruplet X' was computed at line 06. Therefore, $X' = \mathsf{sup}(T)$ where where the set T is equal to $\{view^{\tau_2}[1], \cdots, view^{\tau_2}[n], \langle 1, \mathtt{down}, \mathtt{false}, v_i\rangle\}$. Observe that $\mathsf{sup}(T)$ and $\mathsf{max}(T)$ can differ only in their conflict field. As $\mathsf{sup}(T).cf\ell = X'.cf\ell = \mathtt{false}$, it follows that $X' = \mathsf{sup}(T) = \mathsf{max}(T)$. Consequently, $X' \in view^{\tau_2}$. That is, p_i is not the first process that writes X' in REG, contradiction. Therefore, the write necessarily happens at line 04.

From the precondition at line 04, $view^{\tau_2}$ is $\mathcal{H}(\langle X'.rd - 1, \mathtt{down}, \mathtt{false}, X'.va\ell\rangle)$, and the lemma holds. ◀

▶ **Lemma 15.** *$[(REG^{\tau}$ is $\mathcal{H}(X)) \wedge (X.\ell v\ell = \mathtt{up}) \wedge (\neg X.cf\ell) \wedge (REG^{\tau'}$ is $\mathcal{H}(Y)) \wedge (Y.rd \geq X.rd)] \Rightarrow (Y.va\ell = X.va\ell)$.*

4.5 Proof of the algorithm: using the previous lemmata

▶ **Lemma 16.** *No two processes decide different values.*

Proof. Let r be the smallest round in which a process decides, p_i and $va\ell$ being the deciding process and the decided value, respectively. There is a time τ at which $view_i^{\tau}$ is $\mathcal{H}(\langle r, \mathtt{up}, \mathtt{false}, va\ell\rangle)$. Due to Lemma 15, every homogeneous snapshot starting from round r is necessarily associated with the value $va\ell$. Therefore, only this value can be decided in any round higher than r. Since r was assumed to be the smallest round in which a decision occurs, the consensus agreement property follows. ◀

▶ **Lemma 17.** *For every quadruplet X that is written in REG, $X.va\ell$ is a value proposed by some process.*

▶ **Lemma 18.** *A decided value is a proposed value.*

▶ **Lemma 19.** *Let T be a set of quadruplets. For every $T' \subseteq T$: $\mathsf{sup}(T' \cup \{\mathsf{sup}(T)\}) = \mathsf{sup}(T)$.*

▶ **Lemma 20.** *If there is a time after which a process executes solo, it decides a value.*

▶ **Lemma 21.** *If a single value is proposed, all correct processes decide.*

▶ **Theorem 22.** *The algorithm described in Figure 2 solves the obstruction-free consensus problem (as defined in Section 2.2).*

Proof. The proof follows directly from the Lemma 16 (Agreement), Lemma 18 (Validity), Lemma 20 (OB-Termination), and Lemma 21 (SV-Termination). ◀

5 From Consensus to (n, k)-Set Agreement

The algorithm. Our obstruction-free (n, k)-set agreement algorithm is the same as the one of Figure 2, except that now there are only $m = n - k + 1$ MWMR atomic registers instead of $m = n$. Hence REG is now $REG[1..(n - k + 1)]$.

Its correctness. The arguments for the validity and liveness properties are the same as the ones of the consensus algorithm since they do not depend on the size of REG.

As far as the k-set agreement property is concerned (no more than k different values are decided), we have to show that $(n - k + 1)$ registers are sufficient. To this end, let us consider the $(k - 1)$ first decided values, where the notion "first" is defined with respect to the linearization time of the snapshot invocation (line 02) that immediately precedes the invocation of the corresponding deciding statement (return() at line 03). Let τ be the time just after the linearization of these $(k - 1)$ "deciding" snapshots. Starting from τ, at most $(n - (k - 1)) = (n - k + 1)$ processes access the array REG, which is made up of exactly $(n - k + 1)$ registers. Hence, after time τ, these $(n - k + 1)$ processes execute the consensus algorithm of Figure 2, where $(n - k + 1)$ replaces n, and consequently at most one new value is decided. Therefore, at most k values are decided by the n processes.

6 From One-shot to Repeated (n, k)-Set Agreement

6.1 The repeated (n, k)-set agreement problem

In the repeated (n, k)-set agreement problem, processes executes a sequence of (n, k)-set agreement instances. Hence, a process p_i invokes sequentially the operation propose$(1, v_i)$, then propose$(2, v_i)$, etc., where $sn_i = 1,\ 2,\ \ldots$ is the sequence number of its current instance, and v_i is the value it proposes to this instance.

It would be possible to associate a specific instance of the base algorithm described in Figure 2 with each sequence number, but this would require $(n - k + 1)$ atomic read/write registers per instance. The next section shows that we can solve the repeated problem with only $(n - k + 1)$ atomic registers. According to the complexity results of [10], it follows that this algorithm is optimal in the number of atomic registers it uses, which consequently closes the space complexity discussion regarding repeated (n, k)-set agreement.

6.2 Adapting the algorithm

From quadruplets to sixuplets. Instead of a quadruplet, an atomic read/write register is now a sixuplet $X = \langle sn, rd, \ell v\ell, cf\ell, va\ell, dcd\rangle$. The four fields $X.rd$, $X.\ell v\ell$, $X.cf\ell$, $X.va\ell$ are the same as before. The additional field $X.sn$ contains a sequence number, while the other additional field $X.dcd$ is an initially empty list. From a notational point of view, the jth element of this list is denoted $X.dcd[j]$; it contains a value decided by the jth instance of the repeated (n, k)-set agreement.

The total order on sixuplets "$>$" is the classical lexicographical order defined on the first five fields, while relation "$\sqsupset$" is now defined as follows:

$$X \sqsupset Y \overset{def}{=} (X > Y) \wedge [(X.sn > Y.sn) \vee (X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Y.cf\ell \wedge X.va\ell = Y.va\ell)].$$

Local variables. Each process p_i now manages two local variables whose scope is the whole repeated (n, k)-set agreement problem.

```
operation propose($sn_i, v_i$) is
(01) repeat forever
(02)   $view \leftarrow REG$.snapshot();
(03)   case ($\exists r > 0, va\ell : \forall x : view[x] = \langle sn_i, r, \texttt{up}, \texttt{false}, va\ell, -\rangle$) then
                 $dcd_i[sn_i] \leftarrow va\ell$; return($va\ell$)
(04)        ($\exists r > 0, va\ell : \forall x : view[x] = \langle sn_i, r, \texttt{down}, \texttt{false}, va\ell, -\rangle$) then
                 $REG$.write($1, \langle sn_i, r+1, \texttt{up}, \texttt{false}, va\ell, dcd_i\rangle$)
(05)        ($\exists r > 0, va\ell, \ell evel : \forall x : view[x] = \langle sn_i, r, \ell evel, \texttt{true}, va\ell, -\rangle$) then
                 $REG$.write($1, \langle sn_i, r+1, \texttt{down}, \texttt{false}, va\ell, dcd_i\rangle$)
(06)        otherwise let $\langle inst, r, \ell evel, conflict, va\ell, dec\rangle$
                          $\leftarrow$ sup($view[1], \cdots, view[n], \langle sn_i, 1, \texttt{down}, \texttt{false}, v_i, dcd_i\rangle$);
(07)                 if ($inst > sn_i$) then $dcd_i[sn_i] \leftarrow dec[sn_i]$; return $dcd_i[sn_i]$ end if
(08)                 $x \leftarrow$ smallest index such that $view[x] =$ min($view[1], \cdots, view[n]$);
(09)                 $REG$.write($x, \langle inst, r, \ell evel, conflict, va\ell, dec\rangle$)
(10)   end case
(11) end repeat.
```

Figure 3 Repeated obstruction-free Consensus.

- The variable sn_i, initialized to 0, is used by p_i to generate its sequence numbers. It is assumed that p_i increments sn_i before invoking propose(sn_i, v_i).
- The local list dcd_i is used by p_i to store the value it has decided during the previous instances of the (n,k)-set agreement. Hence, $dcd_i[j]$ contains the value decided by p_i during the jth instance.

The algorithm. The algorithm executed by a process p_i is described in Figure 3. Parts that are new with respect to the base algorithm appear in blue in Figure 2. We detail the internals of our construction below.

- Line 03. When all the entries of the view obtained by p_i contain only sixuplets whose first five fields are equal, p_i decide the value $va\ell$. But before returning $va\ell$, p_i writes $va\ell$ in $dcd_i[sn_i]$. The idea is that, when p_i will execute the next (n,k)-set agreement instance (whose sequence number will be $sn_i + 1$), it will be able to help processes, whose current sequence number sn' are smaller than sn_i.
- Line 04. In this case, p_i obtains a view where the first five entries are equal to $\langle sn_i, r, \texttt{down}, \texttt{false}, va\ell\rangle$. It then writes in $REG[1]$ the value $\langle sn_i, r, \texttt{down}, \texttt{false}, va\ell, dcd_i\rangle$.
- Line 05. Similar to the previous case, except that a conflict now appears in the view of p_i.
- Lines 06-10. Process p_i computes the supremum of the snapshot value $view$ obtained at line 03 plus the sixuplet $\langle sn_i, 1, \texttt{down}, \texttt{false}, va\ell, dcd_i\rangle$. There are two cases to consider.
  - If the sequence number of this supremum $inst$ is greater than sn_i (line 07), p_i can benefit from the list of values already decided in (n,k)-set agreement instances whose sequence number is greater than sn_i. This help is obtained from $dec[sn_i]$. Consequently, similarly to line 03, p_i writes this value in $dcd_i[sn_i]$ and decides it.
  - If now $inst$ equals sn_i, process p_i executes the same operations as in our base algorithm (lines 08–09).

It follows from the algorithm depicted in Figure 3 that solving repeated (n,k)-set agreement in an anonymous system does not require more atomic read/write registers than the base non-repeated version. The only additional cost lies in the size of the atomic registers which contain two supplementary unbounded fields. As pointed out in the introduction, the lower bound established in [10] induces that our solution is space optimal.

```
function sup($T$) is of each of them is now a set of values
(S1')   let $\langle r, \ell evel, cf\ell, valset\rangle$ be max($T$);
(S2')   let $vals(T)$ be $\{v \mid \langle r, -, -, valset\rangle \in T \wedge v \in valset\}$;
(S3)    let $conf\ell ict1(T)$ be $\exists \langle r, -, conflict, -\rangle \in T$;
(S4')   let $conf\ell ict2(T)$ be $|vals(T)| > x$;
(S5)    let $conf\ell ict(T)$ be $conf\ell ict1(T) \vee conf\ell ict2(T)$;
(N)     if $conf\ell ict(T)$ then $vals'(T) \leftarrow valset$
                          else $vals'(T) \leftarrow$ the set of the (at most) $x$ greatest values in $vals(T)$ end if;
(S6')   return$\big(\langle r, \ell evel, conf\ell ict(T), vals'(T)\rangle\big)$.
```

Figure 4 Function sup() suited to x-obstruction-freedom.

7 From Obstruction-Freedom to x-Obstruction-Freedom

This section extends the base algorithm to obtain an algorithm that solves the x-obstruction-free (n, k)-set agreement problem.

Notion of x-obstruction-freedom. This progress condition, introduced in [30], is a natural generalization of obstruction-freedom, which corresponds to the case where $x = 1$. It guarantees that for every set of processes P, with $|P| \leq x$, every correct process in P returns from its operation invocation if no process outside P takes steps for a "long enough" period of time. It is easy to see that x-obstruction-freedom and wait-freedom are equivalent in any n-process system where $x \geq n$. Differently, when $x < n$, x-obstruction-freedom depends on the concurrency pattern while wait-freedom does not.

On the value of x. We assume that $x \leq k$. Such an assumption follows from the impossibility result stating that (n, k)-set agreement cannot be wait-free solved for $n > k$, when any number of processes may crash [4, 18, 27].

Termination. In regard to x-obstruction-freedom, the Validity, Agreement and SV-Termination properties defining obstruction-free (n, k)-set agreement are the same as the ones stated in Section 2.2. The OB-Termination property now becomes:

- x-OB-termination. If there is a time after which at most x correct processes execute concurrently, each of these processes eventually decides a value.

The shared memory. To cope with the x-concurrency allowed by obstruction-freedom, that is the fact that up to x processes may compete, the array REG is such that it has now $m = n - k + x$ entries. At core, this modification in the size of the array comes from the fact that the algorithm terminates in more scenarios than the the ones covered with obstruction-freedom.

Content of a quadruplet. In the base algorithm, the four fields of a quadruplet X are a round number $X.rd$, a level $X.\ell v\ell$, a conflict value $X.cf\ell$, and a value $X.val$. Coping with x-concurrency requires to replace the last field, which was initially a singleton, with a set of values denoted hereafter $X.valset$.

Modifying function sup(). Coping with x-concurrency requires also to adapt function sup(). We describe its novel definition at Figure 4. The lines that are modified (with respect to the base definition) are followed by a "prime". We also add a new line (marked N). In detail, our modifications are the following.

```
operation propose($v_i$) is
(01) $Q \leftarrow \langle 1, \texttt{down}, \texttt{false}, \{v_i\} \rangle$;
(02) repeat forever
(03)     $view \leftarrow REG$.snapshot();
(04)     case $(\exists r > 0, valset : \forall x : view[x] = Q = \langle r, \texttt{up}, \texttt{false}, valset \rangle)$ then
                 return any value in $valset$
(05)          $(\exists r > 0, valset : \forall x : view[x] = Q = \langle r, \texttt{down}, \texttt{false}, valset \rangle)$ then
                 $Q \leftarrow \langle r+1, \texttt{up}, \texttt{false}, valset \rangle$; $REG$.write$(1, Q)$
(06)          $(\exists r > 0, valset, \ell evel : \forall x : view[x] = Q = \langle r, \ell evel, \texttt{true}, valset \rangle)$ then
                 let $v$ be any value in $valset$;
                 $Q \leftarrow \langle r+1, \texttt{down}, \texttt{false}, \{v\} \rangle$;
                 $REG$.write$(1, Q)$;
(07)          otherwise let $Q \leftarrow$ sup$(view[1], \cdots, view[n], Q)$;
(08)                   $x \leftarrow$ smallest index such that $view[x] \neq Q$;
(09)                   $REG$.write$(x, Q)$
(10)     end case
(11) end repeat.
```

Figure 5 Anonymous x-obstruction-free Consensus.

- Line S1'. The last field of a quadruplet is now a set of values, denoted *valset*. Regarding the lexicographical ordering, we order the sets *valset* as follows. We order first by size, then sets of the same size are ordered from their greatest to their smallest element.
- Line S2'. The set $vals(T)$ is now the union of all the *valset* associated with the greatest round number appearing in T.
- Lines S3 and S5: We do not modify these lines.
- Line S4'. $conf\ell ict2(T)$ is modified to take into account x-concurrency. A conflict is now discovered when more than x (instead of 1) values are associated with the round number of the maximal element of T.
- New line N. The set $vals'(T)$ is equal to *valset* if $conflict(T) = \texttt{true}$. Otherwise, it contains the (at most) x greatest values of $vals(T)$.
- Line S6'. The quadruplet returned by sup(T) differs from the one computed in Figure 2, and its last field is now the set $vals'(T)$.

Solving x-Obstruction-free (n, k)-set agreement. Figure 5, describe our x-obstruction-free (n, k)-set agreement solution. We now present its internals, detailing the key differences with our base construction described at Figure 2.

- The relation "$\sqsupset$" introduced in Section 4.1 is extended to take into account the fact that the last field of a quadruplet is now a non-empty set of values. It becomes:

$$X \sqsupset Y \overset{def}{=} (X > Y) \wedge [(X.rd > Y.rd) \vee (X.cf\ell) \vee (\neg Y.cf\ell \wedge X.valset \supseteq Y.valset)].$$

- Each process p_i maintains a local quadruplet denoted Q, and containing the last quadruplet it has computed. Initially, Q is equal to $\langle 1, \texttt{down}, \texttt{false}, \{v_i\} \rangle$ (line 01). This quadruplet allows its owner p_i to have an order on the quadruplets it champions during the execution of propose(v_i): if p_i champions Q at time τ, and champions Q' at time $\tau' \geq \tau$, we have $Q' \sqsupseteq Q$. This is to ensure x-OB-termination.

- The meaning of the three predicates at lines 04-06, is the following. All entries of *view* are the same and are equal to Q, where the content of Q is either $\langle r, \texttt{up}, \texttt{false}, \mathit{valset}\rangle$, or $\langle r, \texttt{down}, \texttt{false}, \mathit{valset}\rangle$, or $\langle r, \ell evel, \texttt{true}, \mathit{valset}\rangle$. Hence, according to the terminology of the proof of our base construction (see Section 4.1), *view* is homogeneous, i.e., *view* is $\mathcal{H}(Q)$, where Q obeys to some predefined pattern.
- Lemma 15 needs to be re-formulated to take into account the last field of a quadruplet. It becomes:

  $$[(REG^{\tau} \text{ is } \mathcal{H}(X)) \wedge (X.\ell v\ell = \texttt{up}) \wedge (\neg X.cf\ell) \wedge (REG^{\tau'} \text{ is } \mathcal{H}(Y)) \wedge (Y.rd \geq X.rd)] \Rightarrow (Y.\mathit{valset} \supseteq X.\mathit{valset} \vee X.\mathit{valset} \supseteq Y.\mathit{valset}).$$

  The lemma is true as soon as the number of participating processes does not exceed the number of available registers in *REG*.
- For the agreement property, we have to show that $(n - k + x)$ registers are sufficient. Our reasoning is similar to the one depicted in Section 5. More precisely, let us consider the $(k - x)$ first decided values, where the notion "first" is defined with respect to the linearization time of the snapshot invocation (line 03) that immediately precedes the invocation of the corresponding deciding statement ($\texttt{return}()$ at line 04). Let τ be the time just after the linearization of these $(k - x)$ "deciding" snapshots. Starting from τ, at most $(n - (k - x)) = (n - k + x)$ processes access the array *REG*, which is made up of exactly $(n - k + x)$ registers. Consider the $(k - x + 1)$-th deciding snapshot, let it be at $\tau' > \tau$. According to the precondition at line 04, $REG^{\tau'}$ is $\mathcal{H}(X)$ for some X with $X.\ell v\ell = \texttt{up}$ and $X.cf\ell = \texttt{false}$. Observe that in such case $|X.\mathit{valset}| \leq x$.

  According to the new statement of Lemma 15, since starting from τ the number of participating processes is always less than the number of registers, then all the deciding snapshots computed after τ' are associated with a set of values that is either a subset or a superset of $X.\mathit{valset}$. Hence, at most x values can be decided starting from τ'.
- Regarding now x-OB-termination, let us first notice that the underlying snapshot algorithm is non-blocking [15]. Hence, it ensures that, whatever the concurrency pattern is, at least one snapshot invocation always terminates. Then, the key is at line 06. When a process p_i detects a conflict ($Q.cf\ell = \texttt{true}$), it starts a new round with a singleton set. Hence, if there is a finite time after which no more than x processes are taking steps, there is a round after which at most x values survive and appear in the next rounds. From that round, no new conflict can appear, and eventually each of the (at most) x running processes obtains a snapshot entailing a decision.

8 Conclusion

In this paper, we first present a one-shot obstruction-free (n, k)-set agreement algorithm for a system made up of n asynchronous anonymous processes that communicate with atomic read/write registers. This algorithm uses only $(n - k + 1)$ registers. In terms of space complexity, it is the best algorithm known so far, and in the case of consensus it is asymptotically optimal [14]. This algorithm answers the challenge posed in [8], and establishes a novel upper bound of $(n - k + 1)$ on the number of registers to solve one-shot obstruction-free (n, k)-set agreement. This upper bound improves the ones stated in [10] for anonymous and non-anonymous systems.

Further, we introduce a simple extension of our base construction that solves repeated (n, k)-set agreement . The lower bound of $(n - k + 1)$ atomic registers was established in [10] for this problem. Our algorithm proves that this bound is tight. Then, we detail a one-shot algorithm to solve the (n, k)-set agreement problem in the context of x-obstruction-freedom. This algorithm makes use of $(n - k + x)$ atomic read/write registers.

All our algorithms rely on the same round-based data structure. The base one-shot algorithm does not require persistent local variables, and in addition to a proposed value, an atomic register solely contains two bits and a round number. The algorithm solving repeated (n, k)-set agreement requires that each atomic register includes two additional fields.

Let us call "MWMR-*nb*" of a problem P, the minimal number of MWMR atomic registers needed to solve P in an asynchronous system of n processes. This paper shows that $(n-k+1)$ is the MWMR-*nb* of repeated obstruction-free (n, k)-set agreement. We conjecture that $(n - k + 1)$ is also the MWMR-*nb* of one-shot obstruction-free (n, k)-set agreement, and more generally that $(n - k + x)$ is the MWMR-*nb* of one-shot x-obstruction-free (n, k)-set agreement, when $1 \leq x \leq k < n$.

References

1 Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890 (1993)

2 Aguilera M., A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT news, DC column*, 35(2):36–59 (2004)

3 Attiya H., Guerraoui R., Hendler D., and Kuznetsov P., The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4), Article 24, 33 pages (2009)

4 Borowsky E. and Gafni E., Generalized FLP impossibility result for t-resilient asynchronous computations. *Proc. 25-th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91–100 (1993)

5 Bouzid Z. and Raynal M. and Sutra P. Anonymous Obstruction-free (n, k)-Set Agreement with $n - k + 1$ Atomic Read/Write Registers *RR 2027, Univerité de Rennes 1*, (2015)

6 Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132–158 (1993)

7 Delporte C., Fauconnier H., Gafni E., and Lamport L., Adaptive register allocation with a linear number of registers. *Proc. 27th Int'l Symposium on Distributed Computing (DISC'13)*, Springer LNCS 8205, pp. 269–283 (2013)

8 Delporte C., Fauconnier H., Gafni E., and Rajsbaum S., Black art: obstruction-free k-set agreement with |MWMR registers| < |proccesses|. *Proc. First Int'l Conference on Networked Systems (NETYS'13)*, Springer LNCS 7853, pp. 28–41 (2013)

9 Delporte C., Fauconnier H., Gafni E., and Rajsbaum S., Linear space bootstrap communication schemes. *Theoretical Computer Science*, 561:122–133 (2015)

10 Delporte C., Fauconnier H., Kuznetsov P. and Ruppert E., On the space complexity of set agreement. *Proc. 34th Int'l Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press (2015)

11 Ellen Fich F., Luchangco V., Moir M., and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer LNCS 3724, pp. 78–92 (2005)

12 Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382 (1985)

13 Flocchini P., Prencipe G., Santoro N., and Widmayer P., Hard tasks for weak robots: the role of common knowledge in pattern formation by autonomous mobile robots. *Proc. 10th Int'l Symposium on Algorithms and Computation (ISAAC'99)*, Springer LNCS 1741, pp. 93–102 (1999)

14 Gelashvili R., Optimal Space Complexity of Consensus for Anonymous Processes *Proc. 29th Int'l Symposium on Distributed Computing (DISC'15)*, Springer LNCS, *in press*, (2015)

15 Guerraoui R. and Ruppert E., Anonymous and fault-tolerant shared-memory computations. *Distributed Computing*, 20:165–177 (2007)

16 Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149 (1991)

17 Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522–529 (2003)

18 Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923 (1999)

19 Herlihy M.P. and Shavit N., *The art of multiprocessor programming.* Morgan Kaufmann, 508 pages (2008) (ISBN 978-0-12-370591-4).

20 Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492 (1990)

21 Lamport L., Concurrent reading while writing. *Communications of the ACM*, 20(11):806–811 (1977)

22 Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77–85 (1986)

23 Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing: vol. 4 of Advances in Comp. Research,* JAI Press, 4:163–183 (1987)

24 Merritt M. and Taubenfeld G., Computing with infinitely many processes. *Information & Computation*, 233:12–31 (2013)

25 Peterson G.L., Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5:46–55 (1983)

26 Raynal M., *Concurrent programming: algorithms, principles, and foundations.* Springer, 530 pages (2013) (ISBN 978-3-642-32026-2).

27 Saks M.S. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal Computing* 29(5):1449–1483 (2000)

28 Suzuki I. and Yamashita M., Distributed anonymous mobile robots. *Proc. 3rd Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'96)*, Carleton University Press, pp. 313–330 (1996)

29 Taubenfeld G., *Synchronization algorithms and concurrent programming.* Pearson Education/Prentice Hall, 423 pages (2006) (ISBN 0-131-97259-6).

30 Taubenfeld G., Contention-sensitive data structure and algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 157–171 (2009)

31 L Zhu, A Tight Space Bound for Consensus. Private communication (2015)

Making “Fast” Atomic Operations Computationally Tractable

Antonio Fernández Anta[*1], **Nicolas Nicolaou**[†2], **and Alexandru Popa**[3]

1 **IMDEA Networks Institute, Madrid, Spain**
`antonio.fernandez@imdea.org`

2 **IMDEA Networks Institute, Madrid, Spain**
`nicolas.nicolaou@imdea.org`

3 **Department of Computer Science, Nazarbayev University, Astana, Kazakhstan**
`alexandru.popa@nu.edu.kz`

Abstract

Communication overhead is the most commonly used performance metric for the operation complexity of distributed algorithms in message-passing environments. However, aside with communication, many distributed operations utilize complex computations to reach their desired outcomes. Therefore, a most accurate operation latency measure should account of both *computation* and *communication* metrics.

In this paper we focus on the efficiency of read and write operations in an *atomic read/write shared memory* emulation in the message-passing environment. We examine the operation complexity of the best known atomic register algorithm, presented in [2], that allows all read and write operations to complete in a *single* communication round-trip. Such operations are called *fast*. At its heart, the algorithm utilizes a predicate to allow processes to compute their outcome. We show that the predicate used in [2] is *computationally hard*, by devising a computationally equivalent problem and reducing that to *Maximum Biclique*, a known *NP-hard* problem. To improve the computational complexity of the algorithm we derive a *new predicate* that leads to a new algorithm, we call ccFast, and has the following properties: (i) can be computed in polynomial time, rendering each read operation in ccFast tractable compared to the read operations in the original algorithm, (ii) the messages used in ccFast are reduced in size, compared to the original algorithm, by almost a linear factor, (iii) allows all operations in ccFast to be *fast*, and (iv) allows ccFast to preserve atomicity. A *linear time* algorithm for the computation of the new predicate is presented along with an analysis of the message complexity of the new algorithm. We believe that the new algorithm redefines the term *fast* capturing both the *communication* and the *computation* metrics of each operation.

1998 ACM Subject Classification C.3.4 Distributed Systems, C.4 Performance of Systems

Keywords and phrases atomicity, read/write objects, shared memory, computational complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.19

[*] The work of the first author is supported in part by FP7-PEOPLE-2013-IEF grant ATOMICDFS No:629088, Ministerio de Economia y Competitividad grant TEC2014- 55713-R, Regional Government of Madrid (CM) grant Cloud4BigData (S2013/ICE-2894, co- funded by FSE & FEDER), NSF of China grant 61520106005, and European Commission H2020 grants ReCred and NOTRE.

[†] The work of the second author is fully supported byFP7-PEOPLE-2013-IEF grant ATOMICDFS No:629088.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 19; pp. 19:1–19:16
Leibniz International Proceedings in Informatics
LIPICS 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 challenging 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) network locations (replica servers). Replication however raises the challenge of consistency, as multiple object copies can be accessed concurrently by multiple processes. To determine the value of the object when this is accessed concurrently, researchers defined several consistency guarantees, the strongest of those being *atomicity*. 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 preceding write operation, and this value is at least as recent as that returned by any preceding read.

The seminal work of Attiya, Bar-Noy, and Dolev [1], was the first to present an algorithm, we refer to as ABD, to implement Single-Writer, Multiple-Reader (SWMR) atomic objects, in message-passing, crash-prone, and asynchronous environments. Here, $\langle timestamp, value\rangle$ pairs are used to order the write operations, and each operation is guaranteed to terminate as long as some majority of replica servers do not crash. The write protocol involves a single round-trip communication, while the read protocol involves two round-trip stages. In particular, the writer increments the timestamp for each write and propagates the new value along with its timestamp to some majority of replicas. The readers are implemented as a two-phase protocol, where the first phase obtains from some majority of the replicas their $\langle timestamp, value\rangle$ pairs, and uses the value corresponding to the highest timestamp as the return value. Before returning, each reader performs a second phase, in which it propagates the highest $\langle timestamp, value\rangle$ pair to some majority of replica servers, ensuring that any subsequent read will discover the value that is at least as recent. Here atomicity is guaranteed in all executions relying on the fact that any two majorities have a non-empty intersection. Avoidance of the second round-trip could lead to violations of atomicity. Following this development, a folklore belief persisted that in asynchronous multi-reader atomic memory implementations “reads must write”.

The work by Dutta et al. [2] refuted this belief, by presenting atomic register implementations where reads involve only a *single* communication round-trip. Such an implementation is called *fast*. This is shown to be possible whenever the number of readers R is appropriately constrained with respect to the number of replicas S and the maximum number of crashes f in the system; this constraint is expressed by $R < \frac{S}{f} - 2$. In this same work the authors showed that it is not possible to devise fast implementations in the multiple-write and multiple-reader (MWMR) model. Subsequently, works like [5, 6], proposed implementations in the SWMR model where some operations were allowed to perform two communication round-trips, in an attempt to relax the constraint proposed in [2] and allow unbounded number of readers. Such implementations traded communication for scalability. Under conditions of low concurrency, this requirement allowed most reads to complete in a single communication round-trip. Even with this relaxed model [6] showed that MWMR implementations where all write are fast are not possible. Following these developments, [3] provided tight bounds on the efficiency of read/write operations in terms of communication round-trips, and introduced the first algorithm to allow some fast read and write operations in the MWMR model.

A trend appeared in the algorithms that aimed for fast operations: algorithms with lower communication rounds demanded higher computation overhead at the processes. The first work to question the computational complexities of the "fast" implementations, and how that affects the performance of the algorithms, was presented by Georgiou et al. [4]. In that paper the authors analyzed the computational complexity of the algorithm presented in [3], the only algorithm that allows both fast reads and writes in the MWMR model, and showed both theoretically and experimentally that the computational overhead of the algorithm was suppressing the communication costs. In particular, the authors expressed the predicate used in the algorithm by a computationally equivalent problem and they showed that such a problem is NP-hard. To improve the complexity of the algorithm presented in [3], they proposed a polynomial *approximation* algorithm.

Contributions. In this paper, we focus in the efficiency of read and write operations in distributed SWMR atomic read/write register implementations. We show that the computation costs have an impact on the best known atomic register implementation, presented in [2], and we propose a *deterministic* solution to improve both the computational and communication burdens of the original algorithm while maintaining fault-tolerance and consistency. Enumerated our contributions are the following:

- We introduce a new problem that is computationally equivalent to the predicate used in [2]. We show that the new problem, and thus the computation of the predicate, is *NP-hard*. For our proof we reduce the new problem to the *Maximal Biclique* problem, which is known to be an NP-hard problem.
- We then devise a revised *fast* implementation, called ccFast, which uses a new polynomial time predicate to determine the value to be returned by each read operation. The idea of the new predicate is to examine the replies received in the first communication round of a read operation and determine *how many* processes witnessed the maximum timestamp among those replies. With the new predicate we reduce the size of each message sent by the replicas, and we prove rigorously that atomicity is preserved.
- Finally, we analyze the operation complexity of ccFast, in terms of *communication*, *computation*, and *message length*. For the computational complexity, we provide a linear time algorithm for the computation of the new predicate. The algorithm utilizes *buckets* to count the number of appearances of each timestamp in the collected replies. We present a complexity analysis of the proposed algorithm and we prove that it correctly computes the predicate of ccFast.

Our results lower the bar of operation latency in SWMR atomic object implementations in the message-passing, asynchronous environment and redefine the term *fast* to capture both the communication and computation overheads of the proposed algorithms.

2 Model

We assume a system consisting of three distinct sets of processes: a single process (the writer) 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 servers may *crash* without any notice.

Each process p of the system can be modeled as an I/O Automaton A_p [11]. The automaton A_p of process p is defined over a set of *states*, $states(A_p)$, and a set of *actions*,

$actions(A)$. There is a state $\sigma_{0,p} \in states(A_p)$ which is the initial state of automaton A_p. An algorithm A is the automaton obtained from the composition of automata A_p, for $p \in \mathcal{I}$. A state $\sigma \in states(A)$ is a vector containing a state for each process $p \in \mathcal{I}$ and the state $\sigma_0 \in states(A)$ is the initial state of the system that contains $\sigma_{0,p}$ for each process $p \in \mathcal{I}$. The set of actions of A is $actions(A) = \bigcup_{p \in \mathcal{I}} actions(A_p)$. An *execution fragment* ϕ of A is an alternate sequence $\sigma_1, \alpha_1, \sigma_2, \ldots, \sigma_{k-1}, \alpha_{k-1}, \sigma_k$ of *states* and *actions*, s.t. $\sigma_i \in states(A)$ and $\alpha_i \in actions(A)$, for $1 \leq i \leq k$. An *execution* is the execution fragment starting with some initial state σ_0 of A. We say that an execution fragment ϕ' *extends* an execution fragment ϕ (or execution), denoted by $\phi \circ \phi'$, if the last state of ϕ is the first state of ϕ'. A triple $\langle \sigma_i, \alpha_{i+1}, \sigma_{i+1} \rangle$ is called a *step* and denotes the transition from state σ_i to state σ_{i+1} as a result of the execution of action α_{i+1}. A process p *crashes* in an execution ξ if the event fail_p appears in ξ; otherwise p is *correct.* Notice that if a process p crashes, then fail_p is the last action of that process in ξ.

In a read/write atomic object implementation each automaton A contains an *invocation* action $\mathsf{read}_{p,O}$ (or $\mathsf{write}(v)_{p,O}$) to *invoke* a read (resp. write) operation π on an object O. Similarly, $\mathsf{read\text{-}ack}(v)_{p,O}$ and $\mathsf{write\text{-}ack}(v)_{p,O}$ are the *response* actions and return the result of the operation π on O. The steps that contain the invocation and response actions, are called invocation and response steps respectively. An operation π is *complete* in an execution ξ, if ξ contains both the invocation and the *matching* response actions 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 action of π appears in ξ. In other words each operation invokes one operation at a time. Finally we say that 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. The 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. Atomicity is defined as follows [9]. For any execution of a memory service, all the completed read and write operations can be partially ordered by an ordering $\prec$, so that the following properties are satisfied:

P1. The partial order is consistent with the external order of invocation and responses, that is, there do not exist operations π_1 and π_2, such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.

P2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

P3. Every read operation returns the value of the last write preceding it in the partial order, and any read operation ordered before all writes returns the initial value of the object.

For the rest of the paper we assume a single register memory system. By composing multiple single register implementations, one may obtain a complete atomic memory [9]. Thus, we omit further mention of object names.

Efficiency Metrics. We are interested in the *complexity* of each read and write operation. The complexity of each operation π is measured from the invocation step of the π to the response step of π. To measure the complexity of an operation π that is invoked by a process p we use the following three metrics: (i) *communication round-trips* , (ii) *computation steps* taken by p during π, and (iii) *message bit complexity* which measures the length of the messages used during π. A communication round-trip (or simply round) is more formally defined in the following definition that appeared in [2, 6, 5]:

▶ **Definition 1.** Process p performs a communication round during operation π if all of the following hold:

1. p sends request messages that are a part of π to a set of processes,
2. any process q that receives a request message from p for operation π, replies without delay, i.e. without waiting for any other messages before replying to π.
3. when process p receives "enough" replies it terminates the round

At the end of a communication round process p may complete π or start a new round. Operation π is *fast* [2] if it completes after its first communication round; an implementation is fast if in each execution all operations are fast.

3 Fastness and its Implications in Atomic Memory Implementations

The algorithm by Dutta et al. in 2004 [2](we refer to it as Fast) was the first to present an atomic register implementation for the message-passing environment where all read and write operations required just a *single* communication round before completing. The same work showed that for any implementation to be fast it must be the case that the number of readers are constrained with respect to the number of servers and server failures in the service by $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$. Fast is using $\langle timestamp, value\rangle$ pairs as in ABD [1] to impose an order on the write operations. The write operation is almost identical to the one round write in [1]: the writer increments its local timestamp, and sends the new timestamp with the value to be written to the majority of the servers. The read operation is much different as it only takes a singe round to complete. To avoid the second round for each read operation, Fast uses two mechanisms: (i) a recording mechanism at the servers, and (ii) a predicate that uses the server recordings at the readers. Essentially each server records all the processes that witness its local timestamp, in a set called *seen*. This set of processes is reset whenever the server learns a new timestamp. The predicate at the readers is the following:

$$\exists \alpha \in [1, \ldots, |\mathcal{R}|+1] \;\wedge\; MS \subset \mathcal{S} \text{ s.t.} \tag{1}$$

$$\forall s \in MS, s.ts = maxTs \;\wedge |MS| \geq |\mathcal{S}| - \alpha f \;\wedge | \bigcap_{s\in MS} s.seen| \geq \alpha \tag{2}$$

Essentially the reader looks at the *seen* sets of the servers that replied, and tries to extract whether "enough" processes witnessed the maximum timestamp. If the predicate holds, the reader returns the value associated with the maximum timestamp. Otherwise it returns the value associated with the previous timestamp. Notice here that the predicate takes in account *which* processes witnessed the latest timestamp as it examines the intersection of the seen sets. Let us now examine what are the complexity costs of Fast in terms of communication, computation and message size. Table 1, presents the comparison of Fast with ABD in all three complexity metrics. Notice that we assume that all three algorithms utilize the same technique to generate timestamps. Thus, we ommit counting the overhead that the timestamp may incur to the complexities presented in Table 1.

Communication Complexity. As previously mentioned, Fast uses one communication round-trip for each read and write operation. That is, each operation sends messages to all the servers and waits replies from a majority. No further communication is required once those replies are received. ABD on the other hand needs one round per write and two rounds per read operation.

Table 1 Communication, Computation, and Message-Bit Complexities of ABD vs Fast vs ccFast. (WR/RR: write/read-rounds, WC/RC: write/read-computation, WB/RB: write/read-message bits)

Algorithm	**WR**	**RR**	**WC**	**RC**	**WB**	**RB**
ABD	1	2	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$O(\lg\lvert V\rvert)$	$O(\lg\lvert V\rvert)$
Fast	1	1	$O(1)$	$O(\lvert\mathcal{S}\rvert^2 \cdot 2^{\lvert\mathcal{S}\rvert})$	$O(\lg\lvert V\rvert)$	$\Theta(\lvert\mathcal{S}\rvert + \lg\lvert V\rvert)$
ccFast	1	1	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$O(\lg\lvert V\rvert)$	$O(\lg\lvert\mathcal{S}\rvert + \lg\lvert V\rvert)$

Computation Complexity. The reduction on the communication rounds had a negative impact on the computational complexity of Fast. The write operation, as also in ABD, terminates once the appropriate number of servers reply, without imposing any further computation. During the read operation the computation complexity of Fast explodes. If we try to examine all possible subsets MS of $\mathcal{S}$, then we obtain $2^{|\mathcal{S}|}$ possibilities. If we restrict this space to include only the subsets with size $|MS| = |\mathcal{S}| - \alpha f$ for all $\alpha \in [1, \ldots, \mathcal{R}]$ (namely $1 \leq |MS| \leq |\mathcal{S}| - f$), then we may examine up to $2^{(|\mathcal{S}|-f)}$ different subsets. Recall also that each *seen* set contains identifiers from the set $\mathcal{R} \cup \{w\}$, and hence at most $|\mathcal{R}| + 1$ elements. To compute the intersection we need to check for each element if it belongs in all the *seen* sets. As MS may include $|\mathcal{S}| - f$ servers (and thus as many *seen* sets) the computation of the intersection may take $(|\mathcal{S}| - f)(|\mathcal{R}| + 1)$ comparisons. As $|\mathcal{R}|$ is bounded by $|\mathcal{S}|$ then the previous quantity is bounded by $O(|\mathcal{S}|^2)$. So that leads to an upper bound of $O(|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|})$. Such complexity may explode the computation time even when the size of $\mathcal{S}$ is small. As however the communication complexity is linear to the size of $\mathcal{S}$ then small set of servers will keep the communication overhead small. In contrast the computation complexity in ABD is bounded by $O(|\mathcal{S}|)$ as the reader parses the replies of at most $|\mathcal{S}|$ servers to detect the maximum timestamp.

Message Bit Complexity. Finally, for each write operation in both Fast and ABD, all servers may send messages containing a value and a timestamp, thus resulting in a bit complexity of $O(\lg |V|)$ per message. The main difference in the message bit complexity lies in the fact that in Fast servers attach the *seen* set along with the $\langle value, timestamp\rangle$ pair for each read operation. To obtain a tight bound we assume that each server sends a *bitmap* indicating whether each client identifier belongs or not in its *seen* set. As each *seen* set may contain up to $|\mathcal{R}| + 1$ identifiers, and since $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 1$, then the bitmap will contain less than or equal to $|\mathcal{S}|$ bits. Hence the length of each message in Fast is bounded by $\Theta(|\mathcal{S}| + \lg |V|)$.

4 Formulation and Hardness of the Predicate in Fast

We formulate the predicate used in Fast by the following computational problem.

▶ **Problem 1.**
Input: *Two sets* $U_1 = \{s_1, s_2, \ldots, s_n\}$, $U_2 = \{p_1, p_2, \ldots, p_k\}$, *where* $\forall s_i \in U_1$, $s_i \subseteq U_2$. *Moreover, we are given two integers* α *and* f *such that* $n - \alpha f \geq 1$.
Goal: *Is there a set* $M \subseteq U_1$ *such that* $|\cap_{s \in M} s| \geq \alpha$ *and* $|M| > n - \alpha f$*?*

It is easy to see the computational equivalence of the above problem and the predicate in Fast: U_1 can be substituted by the set of all the *seen* sets of the servers that replied, M by MS, and U_2 by $\mathcal{R} \cup \{w\}$. We prove that the Problem 1 is NP-hard via a reduction from the decision version of the Maximum Biclique problem defined below. The reduction is similar to the one in [12] for showing that the Maximum k-Intersection Problem is NP-hard.

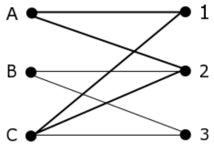


Figure 1 The left side of the graph (nodes A, B and C) corresponds to the elements of the set U_1 and the right side (nodes 1,2 and 3) corresponds to the elements of the set U_2. Thus, $A = \{1, 2\}$, $B = \{2, 3\}$ and $C = \{1, 2, 3\}$. The maximum biclique in this example has two nodes on each side. In the figure, one of the two maximum bicliques is emphasized with bold edges.

▶ **Definition 2** (Maximum Biclique Problem). Given a bipartite graph $G = (X, Y, E)$ a biclique consists of two sets $A \subseteq X$, $B \subseteq Y$ such that $\forall a \in A$, $\forall b \in B$, $(a, b) \in E$. The goal is to decide if the given graph G has a biclique of size at least c.

▶ **Theorem 3.** *Problem 1 is NP-hard.*

Proof. We show that if we can solve Problem 1 in polynomial time, then we can solve the decision version of the Maximum Biclique problem in polynomial time. Given an instance of the biclique problem, i.e., a bipartite graph $G = (X, Y, E)$, we construct the following instance of Problem 1. First, let $U_2 = Y$. Then, each element $s_i \in U_1$ corresponds to a vertex $v \in X$ such that $s_i = \{u \in Y : (v, u) \in E\}$. See Figure 1 for an example.

In order to decide if a biclique of size at least c exists, we solve $|X|$ instances of Problem 1 where α and f are set such that $\alpha \cdot (n - \alpha f) = c$. If there exists a positive instance of Problem 1 among those $|X|$ checked, then there exists a biclique of size at least c. Otherwise, no such biclique exists.

We focus now on two particular values of α and f such that $\alpha \cdot (n - \alpha f) = c$ and we prove the graph G has a biclique of size c with α vertices in the set X and $n - \alpha f$ vertices on the other side, if and only if a subset M that satisfies the constraints of Problem 1 exists.

First, given a biclique $A \cup B$ of size c with $|B| = \alpha$, then the set $M \subseteq U_1$ contains the elements of U_1 associated with the vertices in A. Since the biclique $A \cup B$ has size c, it follows that the number of the sets in M is larger than $c/\alpha = n - \alpha f$.

Conversely, given a set M of size $n - \alpha f$ whose elements have intersection at least α, we can find a biclique of size $c = \alpha \cdot (n - \alpha f)$. The elements $A \subseteq X$ of the biclique are those corresponding to the elements of the set M. Since the elements in the set M have intersection greater than or equal to α, we have that the common neighborhood of the vertices in A is greater than or equal to $\alpha(n - \alpha f)$. Thus, the size of the biclique is at least $c = \alpha \cdot (n - \alpha f)$. ◀

5 Algorithm ccFast: Refining "Fastness" for Atomic Reads

In this section we modify the algorithm presented in [2] to make it even "faster". Since we allow only single round trip operations, the new algorithm adheres to the bound presented in [2] and [3] regarding the possible number of read participants in the service. Thus, the algorithm is possible only if $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 1$. Also, from the results in [2, 6], it follows that such algorithm is impossible in the MWMR model. To expedite the calculation of the predicate we aim to eliminate the use of sets in the predicate and we focused on the question: "Can we preserve atomicity if we know *how many* and not *which* processes read the latest value of a server?". An answer to this question could yield two benefits: (i) reduce the size of messages, and (ii) reduce the computation time of the predicate. We provide a positive answer to this question and we present a new algorithm, we call ccFast, that is communicationally the same and computationally faster than algorithm Fast.

Algorithm 1 Read, Write and Server protocols of algorithm ccFast

```
 1: at the writer $w$
 2: Components:
 3: $ts \in \mathbb{N}^+,\ v, vp \in V, wcounter \in \mathbb{N}^+$
 4: Initialization:
 5: $ts \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, wcounter \leftarrow 0$
 6: function WRITE($val$)
 7:     $vp \leftarrow v; v \leftarrow val;$
 8:     $ts \leftarrow ts + 1$
 9:     $wcounter \leftarrow wcounter + 1$
10:     send($\langle ts, v, vp \rangle, w, wcounter$) to all servers
11:     wait until $|\mathcal{S}| - f$ servers reply
12:     return(OK)
13: end function

14: at each reader $r_i$
15: Components:
16: $ts \in \mathbb{N}^+,\ maxTS \in \mathbb{N}^+,\ v, vp \in V, rcounter \in \mathbb{N}^+$
17: $srvAck \subseteq \mathcal{S} \times M,\ maxTSmsg \subseteq M$
18: Initialization:
19: $ts \leftarrow 0,\ maxTS \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, rcounter \leftarrow 0$
20: $srvAck \leftarrow \emptyset,\ maxTSmsg \leftarrow \emptyset$
21: function READ
22:     $rcounter \leftarrow rcounter + 1$
23:     send($\langle ts, v, vp \rangle, r_i, rcounter$) to all servers
24:     wait until $|srvAck| = |\mathcal{S}| - f$ servers reply //Collect the ($serverid, \langle\langle ts', v', vp' \rangle, views \rangle$) pairs in $srvAck$
25:     $maxTS \leftarrow \max(\{m.ts' | (s, m) \in srvAck\})$
26:     $maxAck \leftarrow \{(s, m) | (s, m) \in srvAck \ \wedge \ m.ts' = maxTS\}$
27:     $\langle ts, v, vp \rangle \leftarrow m.\langle ts', v', vp' \rangle$ for $(*, m) \in maxAck$
28:     if $\exists \alpha \in [1, |\mathcal{R}| + 1]$ s.t. $MS = \{s : (s, m) \in maxAck \ \wedge \ m.views \geq \alpha\}$ and $|MS| \geq |\mathcal{S}| - \alpha f$ then
29:         return($v$)
30:     else
31:         retutn($vp$)
32:     end if
33: end function

34: at each server $s_i$
35: Components:
36: $ts \in \mathbb{N}^+, seen \subseteq \mathcal{R} \cup \{w\},\ v, vp \in V, Counter[1 \ldots |\mathcal{R}| + 1]$
37: Initialization:
38: $ts \leftarrow 0,\ seen \leftarrow \emptyset,\ v, vp \in V, Counter[i] \leftarrow 0$ for $i \in \mathcal{R} \cup \{w\}$
39: function RCV($\langle ts', v', vp' \rangle, q, counter$)          //Called upon reception of a message
40:     if $Counter[q] < counter$ then
41:         if $ts' > ts$ then
42:             $\langle ts, v, vp \rangle \leftarrow \langle ts', v', vp' \rangle$
43:             $seen \leftarrow \{q\}$
44:         else
45:             $seen \leftarrow seen \cup \{q\}$
46:         end if
47:         send($\langle ts, v, vp \rangle,\ |seen|$) to $q$
48:     end if
49: end function
```

The formal specification of the algorithm appears in Figure 1. Here we present a high level description of each protocol in the algorithm. The *counter* variables used throughout the algorithm are solely used to help processes identify “fresh” from “stale” messages due to asynchrony. In the rest of the description we will not refer to the counters, but rather we assume that the messages received by each process are fresh messages.

Write Protocol. To perform a write operation, the writer process w calls the write(val) function. During the write operation the writer stores the value to be written in a variable v and the previous written value in a variable vp (Line 7). Then it increments its local timestamp variable ts (Line 8), and sends a write request along with the triple $\langle ts, v, vp \rangle$ to all the servers and waits for $|\mathcal{S}| - f$ replies. Once those replies are received the operation terminates.

Server Protocol. We now describe the server protocol before proceeding to the read protocol, as it contains the recording mechanism which generates information that is used by each read to determine the value of the register. Each server in $\mathcal{S}$ maintains a timestamp variable along with the values associated with that timestamp. In addition, the server maintains a set of reader and writer identifiers, called *seen*. Initially each server is waiting for read and/or write requests. When a request is received the server examines if the timestamp ts' attached in the request is larger than its local timestamp ts (Line 41). If $ts' > ts$, the server updates its local timestamp and values to be equal to the ones attached in the received message (Line 42), and resets its *seen* set to include only the identifier of the process that sent this message (Line 43); otherwise the server just inserts the identifier of the sender in the *seen* set (Line 45). Then, the server replies to the sender by sending its local $\langle ts, v, vp\rangle$ triple, and the size of its recording set $|seen|$. This is a departure from the Fast algorithm where the server was attaching the complete *seen* set.

Read Protocol. The read protocol is the most involved. When a reader process invokes a read operation it sends read requests along with its local $\langle ts, v, vp\rangle$ triple to all the servers, and waits for $|\mathcal{S}| - f$ of them to reply. Once the reader receives those replies it: (i) discovers the maximum timestamp, $maxTS$, among the messages, (ii) collects all the messages that contained $maxTS$ in a set $maxAck$, and (iii) updates its local $\langle ts, v, vp\rangle$ triple to be equal to the triple attached in one of those messages (Lines 25-27). Then it runs the following predicate on the set $maxAck$ (Line 28):

$$\exists\alpha \in [1, |\mathcal{R}|+1] \text{ s.t. } MS = \{s : (s,m) \in maxAck \wedge m.views \geq \alpha\} \text{ and } |MS| \geq |\mathcal{S}| - \alpha f\,.$$

The predicate examines *how many* processes the maximum timestamp has been sent to. If more than $|\mathcal{S}| - \alpha f$ servers sent this timestamp to more than α processes, for α between $[1, \ldots, |\mathcal{R}|+1]$, then the predicate is true and the read operation returns the value associated with $maxTS$, namely v; otherwise the read operation returns the value associated with $maxTS - 1$, namely vp.

Idea of the predicate. The goal of the predicate is to help a read operation to predict the value that was potentially returned by a preceding read operation. To understand the idea behind the predicate consider the following execution, ξ_1. Let the writer perform a write operation ω and receive replies from a set $\mathcal{S}_1$ of $|\mathcal{S}| - f$ servers. Let a reader follow and perform a read operation ρ_1 that receives replies from a set of servers $\mathcal{S}_2$ again of size $|\mathcal{S}| - f$ that *misses* f servers that replied to the write operation. Due to asynchrony, an operation may *miss* a set of servers if the messages of the operation are delayed to reach any servers in that set. So the two sets intersect in $|\mathcal{S}_1 \cap \mathcal{S}_2| = |\mathcal{S}| - 2f$ servers. Consider now ξ_2 where the write operation ω is not complete and only the servers in $\mathcal{S}_1 \cap \mathcal{S}_2$ receive the write requests. If ρ_1 receive replies from the same set $\mathcal{S}_2$ in ξ_2 then it won't be able to distinguish the two executions. In ξ_1 however the read has to return the value written, as the write in that execution proceeds the read operation. Thus, in ξ_2 the read has to return the value written as well. If we extend ξ_2 by another read operation ρ_2 from a third process, then it may receive replies from a set $\mathcal{S}_3$ missing f servers in $|\mathcal{S}_1 \cap \mathcal{S}_2|$. Thus it may see the value written in $|\mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3| = |\mathcal{S}| - 3f$ servers. But since there is another read that saw the value from these servers (ρ_1) then ρ_2 has to return the written value to preserve atomicity. Observe now that ρ_1 saw the written value from $|\mathcal{S}| - 2f$ servers and each server replied to both $\{w, \rho_1\}$, and ρ_2 saw the written value from $|\mathcal{S}| - 3f$ and each server replied to all three $\{\omega, \rho_1, \rho_2\}$. By continuing with the same logic, we derive the predicate that if a read sees a value written in $|\mathcal{S}| - \alpha f$ servers and each of those servers sent this value to α other processes then we return the written value.

Notice that in order for an operation to see the written value it must be the case that there is at least one server that replied with that value, and thus $|\mathcal{S}| - \alpha f \geq f$. Solving this equation results in $\alpha \leq \frac{\mathcal{S}-1}{f}$. But α is the number of processes in the system. As the maximum number of processes is $|\mathcal{R}| + 1$, hence we derive the bound on the number of possible reader participants that $|\mathcal{R}| < \frac{\mathcal{S}-1}{f}$.

5.1 Algorithm Correctness

To show that the algorithm is correct we need to show that each correct process terminates (liveness) and that the algorithm satisfies the properties of atomicity (safety). As the main departure of ccFast from Fast, is the predicate logic, some of the proofs that follow are very similar to the ones presented in [2]. The lack of complete knowledge of *which* processes witnessed a value, introduced challenges in proving that consistency is preserved even when we know *how many* witnessed a value. Termination is trivially satisfied with respect to our failure model: up to f servers may fail and each operation waits for no more than $|\mathcal{S}| - f$ replies. The atomicity properties can be expressed in terms of timestamps as follows:

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. $\omega(ts') \rightarrow \rho$, or $\omega(ts')$ is concurrent with ρ.

A4. If ρ_1 and ρ_2 are two read operations such that $\rho_1 \rightarrow \rho_2$ and ρ_1 returns ts_1, then ρ_2 returns $ts_2 \geq ts_1$.

Monotonicity allows the ordering of the values according to their associated timestamps. So Lemma 4 shows that the ts variable maintained by each process in the system is monotonically increasing. Let us first make the following observation:

▶ **Lemma 4.** *In any execution ξ of the algorithm, if a server s replies with a timestamp ts at time T, then s replies with a timestamp $ts' \geq ts$ at any time $T' > T$.*

Proof. A server attaches in each reply its local timestamp. Its local timestamp in turn is updated only whenever the server receives a higher timestamp (Lines 37-38). So the server local timestamp is monotonically non-decreasing and the lemma follows. ◀

The following is also true for a server process.

▶ **Lemma 5.** *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 $T' > T$.*

Proof. If the local timestamp of the server s, ts_s, is smaller than ts, then $ts_s = ts$. Otherwise ts_s does not change and remains $ts_s \geq ts$. In any case s replies with a timestamp $ts_s \geq ts$ to π. By Lemma 4 the server s attaches a timestamp $ts' \geq ts_s$, and hence $ts' \geq ts$ to any subsequent reply. ◀

Now we show that the timestamp is monotonically non-decreasing for the writer and the reader processes.

▶ **Lemma 6.** *In any execution ξ of the algorithm, the variable ts stored in any process is non-negative and monotonically non-decreasing.*

Proof. The lemma holds for the writer as it changes its local timestamp by incrementing it every time it performs a write operation. The timestamp at each reader becomes equal to the largest timestamp the reader discovers from the server replies. So it suffices to show that in any two subsequent read from the same reader, say ρ_1, ρ_2 s.t. $\rho_1 \to \rho_2$, then ρ_2 returns a ts' that is bigger or equal to the timestamp ts returned by ρ_1. This can be easily shown by the fact that ρ_2 attaches the maximum timestamp discovered by the reader before the execution of ρ_2. Say this is ts discovered during ρ_1. By Lemma 5 any server that will receive the message from ρ_2 will reply with a timestamp $ts_s \geq ts$. So ρ_2 will discover a maximum timestamp $ts' \geq ts$. If $ts' = ts$ then the predicate will hold for $\alpha = 1$ for ρ_2 and thus it stores $ts' = ts$. If $ts' > ts$ then ρ_2 stores either ts' or $ts' - 1$. In either case it stores a timestamp greater or equal to ts and the lemma follows. ◀

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.

▶ **Lemma 7.** *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 $ts' \geq ts$.*

Proof. According to the algorithm, the write operation ω communicates with a set of $|S_w| = |\mathcal{S}| - f$ servers before completing. Let $|S_1| = |\mathcal{S}| - f$ be the number of servers that replied to the read operation ρ. The intersection of the two sets is $|S_w \cap S_1| \geq |\mathcal{S}| - 2f$ and since $f < |\mathcal{S}|/2$ there exists at least a single server s that replied to both operations. Each server $s \in S_w \cap S_1$ replies to ω before replying to ρ. Thus, by Lemma 5 and since s receives the message from ω before replying to any of the two operations, then it replies to ρ with a timestamp $ts_s \geq ts$. Thus there are two cases to investigate on the timestamp: (1) $ts_s > ts$, and (2) $ts_s = ts$.

Case 1: In the case where $ts_s > ts$, ρ will observe a maximum timestamp $maxTS \geq ts_s$. Since ρ returns either $ts' = maxTS$ of $ts' = maxTS - 1$, then $ts' \geq ts_s - 1$. Thus, $ts' \geq ts$ as desired.

Case 2: In this case all the servers in $S_w \cap S_1$ reply with a timestamp $ts_s = ts$. The read ρ may observe a maximum timestamp $maxTS \geq ts_s$. If $maxTS > ts_s$, then, with similar reasoning as in Case 1, we can show that ρ returns $ts' \geq ts$. So it remains to investigate the case where $maxTS = ts_s = ts$. In this case, at least $|S_w \cap S_1| = |\mathcal{S}| - 2f$ servers replied with $maxTS$ to ρ. Also for each $s \in S_w \cap S_1$, s included both the writer identifier w and r_1 before replying to ω and ρ_2 respectively. So s replied with a size at least $s.views \geq 2$ to ρ_2. Thus, given that $|\mathcal{R}| \geq 2$, the predicate holds for $\alpha = 2$ and the set $S_w \cap S_1$ for ρ, and hence it returns a timestamp $ts' = ts$. And the lemma follows. ◀

So now it remains to show that in two succeeding read operations, the latest operation returns a value that is the same or greater than the value returned by the first read. More formally:

▶ **Lemma 8.** *In any execution ξ of the algorithm, 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. Let the two operations ρ_1 and ρ_2 be executed from the same process, say r_1. As explained in Lemma 6, ρ_2 will discover a maximum timestamp $maxTS \geq ts_1$. If $maxTS > ts_1$, then ρ_2 returns either $ts_2 = maxTS$ or $ts_2 = maxTS - 1$, and thus in both cases

$ts_2 \geq ts_1$. It remains to examine the case where $maxTS = ts_1$. Since $\rho_1 \rightarrow \rho_2$, then any message sent during ρ_2 contains timestamp ts_1. By Lemma 5, every server s that receives the message from ρ_2 replies with a timestamp $ts_s \geq ts_1$. Since $maxTS = ts_1$, then it follows that all $|\mathcal{S}| - f$ servers that replied to ρ_2, sent the timestamp ts_1. Before each server replies adds r_1 in their seen set. So they include a $views \geq 1$ in their messages. Thus, the predicate holds for ρ_2 for $\alpha = 1$ and returns $ts_2 = maxTS = ts_1$.

For the rest of the proof we assume that the read operations are invoked from two different processes r_1 and r_2 respectively. Let $maxTS_1$ be the maximum timestamp discovered by ts_1. We have two cases to consider: (1) ρ_1 returns $ts_1 = maxTS_1 - 1$, or (2) ρ_1 returns $ts_1 = maxTS_1$.

Case 1: In this case ρ_1 returns $ts_1 = maxTS_1 - 1$. It follows that there is a server s that replied to ρ_1 with a timestamp $maxTS_1$. This means that the writer invoked the write operation that tries to write a value with timestamp $maxTS_1$. Since the single writer invokes a single operation at a time (by *well-formedness*), it must be the case that the writer completed writing timestamp $maxTS_1 - 1$ before the completion of ρ_1. Let that write operation be ω. Since, $\rho_1 \rightarrow \rho_2$, then it must be the case that $\omega \rightarrow \rho_2$ as well. So by Lemma 7, ρ_2 returns a timestamp ts_2 greater or equal to the timestamp written by ω, and thus $ts_2 \geq maxTS_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Case 2: This is the case where ρ_1 returns $ts_1 = maxTS_1$. So it follows that the predicate is satisfied for ρ_1, and hence $\exists \alpha \in [1, \ldots, |\mathcal{R}|]$ and a set of servers M_1 such that every server $s \in M_1$ replied with the maximum timestamp $maxTS_1$ and a seen set size $s.views \geq \alpha$, and $|M_1| \geq |\mathcal{S}| - \alpha f$. We know that ρ_2 receives replies from a set of servers $|S_2| = |\mathcal{S}| - f$ before completing. Let M_2 be the set of servers that replied to ρ_2 with a maximum timestamp $maxTS_2$. Since $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$, then

$$|M_1| > |\mathcal{S}| - (\frac{|\mathcal{S}|}{f} - 2)f \Rightarrow |M_1| > f\,.$$

Hence, $S_2 \cap M_1 \neq \emptyset$ and by Lemma 5 every server $s \in S_2 \cap M_1$ replies to ρ_2 with a timestamp $ts_s \geq maxTS_1$. Therefore $maxTS_2 \geq maxTS_1$. If $maxTS_2 > maxTS_1$, then ρ_2 returns a timestamp $ts_2 \geq maxTS_2 - 1 \Rightarrow ts_2 \geq maxTS_1$ and hence $ts_2 \geq ts_1$.

It remains to investigate the case where $maxTS_2 = maxTS_1$. Notice that any server in $s \in S_2 \cap M_1$ is also in M_2. Since ρ_2 may skip f servers that reply to ρ_1, then $|M_1 \cap M_2| \geq |\mathcal{S}| - (a+1)f$. Recall that for each server $s \in M_1 \cap M_2$, s replied with a size $s.views \geq a$ to ρ_1. Also s adds r_2 in its seen set before replying to ρ_2. So there are two subcases to examine: (a) either r_2 was already in the seen set of s, or (b) r_2 was not a member of $s.seen$.

Case 2(a): If r_2 was already a part of the *seen* set of s, then the size of the set remains the same. It also means that r_2 obtained $maxTS_1$ from s in a previous read operation, say ρ_2' from r_2. Since each process satisfies well-formdness, it must be the case that r_2 completed ρ_2' before invoking ρ_2. All the messages sent by ρ_2 contained $maxTS_1$. So by Lemma 5 any server $s \in S_2$ replies to r_2 with a timestamp $ts_s = maxTS_2 = maxTS_1$. In this case $|\mathcal{S}| - f$ servers replied with $maxTS_2$ and their seen set contains at least r_2, having $s.views \geq 1$. Thus, the predicate is valid with $\alpha = 1$ for ρ_2 which returns $ts_2 = maxTS_2 = maxTS_1 = ts_1$.

Case 2(b): This case may arise if r_2 is not part of the seen set of every server $s \in M_1 \cap M_2$. If r_2 is part of the seen set of some server $s' \in M_1 \cap M_2$, then this is resolved by case 2(a).

So each server $s \in M_1 \cap M_2$ inserts r_2 in their seen sets before replying to ρ_2. So if the size of the set $s.views = \alpha$ when s replied to ρ_1, s includes a size $s.views \geq a + 1$ when replying to ρ_2. Notice here that if $\alpha = |\mathcal{R}| + 1$ for ρ_1, then it means that r_2 was already part of the seen set of s when s replied to ρ_1. This case is similar to 2(a). So we assume that $\alpha < |\mathcal{R}| + 1$, in which case $\alpha + 1 \leq |\mathcal{R}| + 1$. Since every server $s \in M_1 \cap M_2$ replies with $s.views \geq \alpha + 1$ to ρ_2 and since $|M_1 \cap M_2| \geq |\mathcal{S}| - (\alpha + 1)f$, then the predicate holds for $\alpha + 1 \leq |\mathcal{R}| + 1$ and the set $MS = M_1 \cap M_2$ for ρ_2, and thus ρ_2 returns $ts_2 = maxTS_2 = maxTS_1 = ts_1$ in this case as well. And this completes our proof. ◀

6 A Linear Algorithm for the Predicate and Complexity of ccFast

Table 1 presents the comparison of the complexities of ccFast with the complexities of both algorithms ABD and Fast.

Communication Complexity. The *communication complexity* of ccFast is identical to the communication complexity of Fast: both read and write operations terminate at the end of their first communication round trip.

Message Bit Complexity. Each message sent in ccFast contains a triple with timestamp and two values. Omitting the timestamp as discussed earlier, then the values alone result in an upper bound of $O(\lg |V|)$ bits. Additionally, each server attaches the size of its *seen* set, which may include $|\mathcal{R}| + 1$ processes. The number of readers however, is bounded by $|\mathcal{S}|$, and hence the size of the seen set can be obtained with $\lg |\mathcal{S}|$ bits. Thus, the size of each message sent in ccFast is bounded by $O(\lg |V| + \lg |\mathcal{S}|)$ bits.

Computation Complexity. Computation is minimal at the writer and server protocols. The most computationally intensive procedure is the computation of the predicate during a read operation. To analyze the *computation complexity* of ccFast we design and analyze an algorithm to compute the predicate during any read operation.

Algorithm 2 presents the formal specification of the algorithm. Briefly, we assume that the input of the algorithm is a set $srvAck$ and a value $maxTS$ which indicate the servers that reply to a read operation and the maximum timestamp discovered among the replies. The algorithm uses a set of $|\mathcal{R}| + 1$ "buckets" each of which is initialized to 0. Running through the set of replies, $srvAck$, a bucket k is incremented whenever a server replied with the maximum timestamp and reports that this timestamp is seen by k processes (Lines 3-7). At the end of the parsing of the $srvAck$ set, each bucket k holds how many servers reported the maximum timestamp and they sent this timestamp to k processes. Once we accumulate this information we check if the number of servers collected in a bucket k are more than $|\mathcal{S}| - kf$. If they are, the procedure terminates returning TRUE; else the number of servers in bucket k is added to the number of servers of bucket $k - 1$ and we repeat the check of the condition (Lines 8-14). At this point the number kept at bucket $k - 1$ indicates the total number of servers that reported that their timestamp was seen by *more or equal* to $k - 1$ processes. This procedure continues until the above condition is satisfied or we reach the smallest bucket. If none of the buckets satisfies the condition the procedure returns FALSE.

▶ **Theorem 9.** *Algorithm 2 implements the predicate used in every read operation in algorithm* ccFast.

Algorithm 2 Linear Algorithm for Predicate Computation.

```
function isValidPredicate($srvAck, maxTS$)
    $buckets \leftarrow Array[1 \ldots |\mathcal{R}| + 1]$, initially $[0, \ldots, 0]$
    for all $s \in srvAck$ do
        if $s.ts == maxTS$ then
            $buckets[s.views] + +$
        end if
    end for
    for $\alpha = |\mathcal{R}| + 1$ to 2 do
        if $buckets[\alpha] \geq (|\mathcal{S}| - \alpha f)$ then
            return TRUE
        else
            $buckets[\alpha - 1] \leftarrow buckets[\alpha - 1] + buckets[\alpha]$
        end if
    end for
    if $buckets[1] == (|\mathcal{S}| - f)$ then
        return TRUE
    end if
    return FALSE
end function
```

Proof. To show that Algorithm 2 correctly implements the predicate used by the read operations in ccFast, we need to show that it returns TRUE whenever the predicate holds and returns FALSE otherwise. Recall that the predicate is the following:

$$\exists \alpha \in [1, |\mathcal{R}|+1] \text{ s.t. } MS = \{s : (s, m) \in maxAck \ \wedge \ m.views \geq \alpha\} \text{ and } |MS| \geq |\mathcal{S}| - \alpha f \,.$$

According to our implementation we have a bucket for each α. For each α the predicate demands that we collect all the servers that replied with $maxTS$ and with $views \geq \alpha$ (set MS). Then we check if these servers are more than $|\mathcal{S}| - \alpha f$. Let $\mathcal{S}_i = \{s : s \in srvAck \ \wedge \ s.ts = maxTS \ \wedge \ s.views = i\}$, for $1 \leq i \leq |\mathcal{R}| + 1$, be the set of servers who replied with $views = i$. Since each server includes a single $views$ number, notice that for any $i, j \in [1, |\mathcal{R}| + 1]$, $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$.

It is easy to see that initially each bucket k, for $1 \leq k \leq |\mathcal{R}| + 1$, holds the number of servers with exactly k views, and hence $bucket[k] = |\mathcal{S}_k|$. Notice that the last bucket $|\mathcal{R}| + 1$ collects all the servers that replied to all possible processes (including the writer). Thus, no server may reply with $views > |\mathcal{R}| + 1$. So, if the predicate is valid for $\alpha = |\mathcal{R}| + 1$, it follows that $MS = S_{|\mathcal{R}|+1}$, and hence $|S_{|\mathcal{R}|+1}| \geq |\mathcal{S}| - (|\mathcal{R}| + 1)f$. Since $bucket[|\mathcal{R}| + 1] = |S_{|\mathcal{R}|+1}|$, then $bucket[|\mathcal{R}| + 1] \geq |\mathcal{S}| - (|\mathcal{R}| + 1)f$ and the condition of Algorithm 2 also holds. Thus, the algorithm returns TRUE in this case.

It remains to investigate any case where $\alpha < |\mathcal{R}| + 1$. Notice that the MS set in the predicate includes all the servers that replied with $views \geq \alpha$. Thus, for any $\alpha < |\mathcal{R}| + 1$,

$$MS = \bigcup_{\alpha \leq i \leq |\mathcal{R}|+1} \mathcal{S}_i \,.$$

Since no two sets $\mathcal{S}_i$ and $\mathcal{S}_j$ intersect, then

$$|MS| = \sum_{\alpha \leq i \leq |\mathcal{R}|+1} |\mathcal{S}_i| \,.$$

When a bucket $k < |\mathcal{R}| + 1$ is investigated the value of the bucket becomes

$$bucket[k] = \sum_{k \leq i \leq |\mathcal{R}|+1} bucket[i]$$

where $bucket[i] = |\mathcal{S}_i|$, the initial value of the bucket. Thus, the above summation can be written as

$$bucket[k] = \sum_{k \leq i \leq |\mathcal{R}|+1} |\mathcal{S}_i| \,.$$

Therefore, $bucket[k] = |MS|$, whenever $k = \alpha$. Hence, if $|MS| \geq |\mathcal{S}| - \alpha f$ in the predicate it must be the case that $bucket[\alpha] \geq |\mathcal{S}| - \alpha f$ in the algorithm. It follows that if the predicate is valid the algorithm returns TRUE. Similarly, if the condition does not hold for the predicate it does not hold for the algorithm either. If there is no α to satisfy the predicate then there is no k to satisfy the condition in the algorithm. Thus, the algorithm in this case returns FALSE, completing the proof. ◀

Finally we can analyze the complexity of Algorithm 2 which in turn specifies the computational complexity of the ccFast. Algorithm 2 traverses once the set *srvAck* and once the array of $|\mathcal{R}| + 1$ buckets. Since, the set *srvAck* may contain at most $|\mathcal{S}|$ servers, and $|\mathcal{R}|$ is bounded by $|\mathcal{S}|$, then the complexity of the algorithm is:

▶ **Theorem 10.** *Algorithm 2 takes $O(|\mathcal{S}|)$ time.*

This shows that we can compute the predicate of algorithm ccFast in *linear* time with respect to the number of servers in the system. This is a huge improvement over the time required by the Fast algorithm, and matches the computational efficiency of the two round ABD algorithm. This result demonstrates that fastness does not necessarily has to sacrifice computation efficiency.

7 Conclusions

In this paper we questioned the overall complexity of algorithms that implement atomic SWMR R/W registers in the asynchronous, message-passing environment where processes are prone to crashes. Communication used to be the prominent operation efficiency metric for such implementations. We pick the best known (in terms of communication) algorithm that implements an atomic SWMR R/W register, Fast, that allows both reads and writes to terminate in just a *single* communication round. We show that the predicate utilized by the Fast to achieve such performance is hard to be computed, and hence the problem is not tractable. Next we present a new predicate that provides the following properties: (i) can be computed in polynomial time, (ii) allows operations to complete in a *single* communication round, and (iii) allows algorithm ccFast to preserve atomicity. A rigorous proof of the correctness of the algorithm is presented. Finally we conclude with a *linear time* algorithm to compute the newly proposed predicate. We believe that the new results redefine the term *fast* in atomic register implementations as operation performance accounts of all, *communication*, *computation*, and *message bit* complexity metrics. It is yet to be determined if the new operation efficiency is optimal or can be further improved.

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 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)*, pages 236–245, 2004.

3 Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)*, pages 240–254, 2009.

4 Chryssis Georgiou, Nicolas Nicolaou, Alexander Russel, and Alexander A. Shvartsman. Towards feasible implementations of low-latency multi-writer atomic registers. In *10th Annual IEEE International Symposium on Network Computing and Applications*, August 2011.

5 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.

6 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.

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 Nancy A. Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.

12 Eduardo C. Xavier. A note on a maximum k-subset intersection problem. *Information Processing Letters*, 112(12):471–472, 2012. doi:10.1016/j.ipl.2012.03.007.

Robust Shared Objects for Non-Volatile Main Memory

Ryan Berryhill[1], Wojciech Golab[2], and Mahesh Tripunitara[3]

1 **University of Toronto, Toronto, Canada**
`ryan@eecg.utoronto.ca`
2 **University of Waterloo, Waterloo, Canada**
`wgolab@uwaterloo.ca`
3 **University of Waterloo, Waterloo, Canada**
`tripunit@uwaterloo.ca`

Abstract

Research in concurrent in-memory data structures has focused almost exclusively on models where processes are either reliable, or may fail by crashing permanently. The case where processes may recover from failures has received little attention because recovery from conventional volatile memory is impossible in the event of a system crash, during which both the state of main memory and the private states of processes are lost. Future hardware architectures are likely to include various forms of non-volatile random access memory (NVRAM), creating new opportunities to design robust main memory data structures that can recover from system crashes. In this paper we advance the theoretical foundations of such data structures in two ways. First, we review several known variations of Herlihy and Wing's linearizability property that were proposed in the context of message passing systems but also apply in our NVRAM-based model, we discuss the limitations of these properties with respect to our specific goals, and we propose an alternative correctness condition called *recoverable linearizability*. Second, we discuss techniques for implementing shared objects that satisfy such properties with a focus on wait-free implementations. Specifically, we demonstrate how to achieve different variations of linearizability in our model by transforming two classic wait-free constructions.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases non-volatile main memory, concurrency, recovery, data structures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.20

1 Introduction

Shared data structures are essential building blocks for modern operating systems and applications, which are empowered almost exclusively by multi-core hardware platforms. Although the multi-core revolution has propelled research in this area to new heights over the last decade, questions pertaining to specifying and implementing concurrent data structures were considered in the literature long before thread-level parallelism became mainstream. Dijkstra's pioneering work on concurrent programming dates back to 1965 [10], followed by a series of seminal papers on inter-process communication, wait-free synchronization, and linearizability [14, 16, 22, 23]. The fundamental abstractions introduced in this body of work have been studied widely in the context of various theoretical models of shared memory computation that capture precise assumptions regarding the synchrony and reliability of processes, as well as the set of primitive operations available for accessing memory. In particular, recent research has paid close attention to asynchronous models, in which there is

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 20; pp. 20:1–20:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

no bound on the amount of time a process takes to transition to its next step, or to complete a memory operation. This assumption reflects in a meaningful way the behavior of modern memory hierarchies, in which different media (e.g., L1 cache vs. L2 cache vs. main memory) incur vastly different and often unpredictable access latencies, as well as the effect of the operating system (e.g., via preemption and interrupts) on the liveness of processes. Aside from capturing these important aspects of real world performance, asynchrony is also related to reliability in a precise way: algorithms that provide non-blocking progress properties (e.g., lock-freedom and wait-freedom) in an asynchronous environment with reliable processes continue to provide the same progress properties if crash failures are introduced. Informally speaking, this property holds because a process that crashes permanently at an arbitrary point in the execution of its algorithm is indistinguishable to the other processes from one that is merely very slow.

Owing to its simplicity and intimate relationship with asynchrony, the crash failure model is almost ubiquitous in the treatment of non-blocking shared memory algorithms and message passing protocols [4, 7, 11, 14, 20]. In comparison, much less attention has been paid to crash-recovery models, in which a failed process may be resurrected after a crash failure. For example, in the message passing paradigm a process may crash and recover state information either from its private stable storage or from another process on a different node [3]. Although similar techniques are in principle applicable in the shared memory paradigm of computation, they are poorly matched to modern multi-core architectures with volatile SRAM-based caches and DRAM-based main memories [29]. Any state stored in main memory is lost entirely in the event of a system crash or power loss, and recording recovery information in non-volatile secondary storage (e.g., on a hard disk drive or solid state drive) imposes overheads that are unacceptable for performance-critical tasks, such as synchronizing threads inside the operating system kernel.

In this paper, we consider correctness properties and implementation techniques for data structures intended for future shared memory architectures that incorporate *non-volatile random-access memory (NVRAM)* – a form of main memory that promises to marry performance comparable to DRAM with the high density and persistence of secondary storage. We assume that NVRAM is accessed using memory operations (e.g., reads, writes, and read-modify-write primitives), similarly to ordinary DRAM, and that such operations can be made both atomic and durable through appropriate extensions to conventional caching and memory ordering mechanisms [9, 18, 25, 28, 30]. Under these assumptions, concurrent data structures such as stacks, queues, and trees may reside directly in NVRAM and algorithms for accessing them may follow conventional techniques for synchronization. However, conventional techniques do not address the problem of recovering such structures following a failure, such as may occur when a multiprocessor suffers a power outage, leading to the loss of all volatile state including the program counter and other vital CPU registers.

Our technical contributions with respect to robust shared objects for multiprocessors that incorporate NVRAM are the following:

1. We refine the conventional abstract model of a shared memory multiprocessor by introducing non-volatility.
2. We survey known correctness properties proposed for shared objects in models with crash and crash-recovery failures, discuss their limitations, and propose an alternative property we call *recoverable linearizability* (or *R-linearizability* for short).
3. We explore techniques for transforming ordinary linearizable implementations into R-linearizable ones, with a special focus on wait-freedom. Our discussion uses as examples Herlihy's universal wait-free construction [14] and a classic wait-free implementation of MRSW registers from SRSW registers [15].

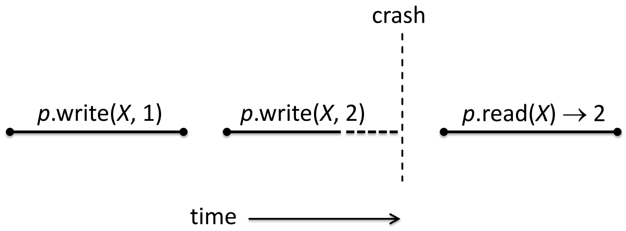


Figure 1 Example execution in which process p writes 1 to object X, then begins writing 2 to X, fails due to a system crash before the write returns, and then reads 2 from X.

2 Related Work

Constructing robust shared objects for NVRAM requires two ingredients: a correctness property that provides meaningful guarantees in the presence of failures, and a set of algorithmic techniques that leverage the non-volatility of the storage medium for recovery. Conventional relational databases provide both, namely serializability for correctness and write-ahead logging for recovery (e.g., ARIES [26]), but perform orders of magnitude slower than main memory data structures owing to their internal complexity as well as their use of a centralized recovery log in secondary storage. Coburn et al. propose NV-Heaps as an alternative method of providing transactional access to persistent shared objects [8]. NV-Heaps use NVRAM directly to store both object state and recovery information in the form of *operation descriptors*, which are similar to recovery logs but more fine-grained. Like conventional databases, NV-Heaps use locks for concurrency control and perform recovery by analyzing the operation descriptors while execution of new transactions is temporarily suspended. Mnemosyne is another transactional interface to NVRAM that uses lock-based concurrency control and log-based recovery [35]. Venkataraman et al. define Consistent and Durable Data Structures (CDDSs), which provide lock-free access to readers but rely on mutual exclusion to synchronize writers [33]. These structures are linearizable in failure-free executions, and any updates that are interrupted by a crash failure are discarded on recovery.

Aside from techniques targeted specifically at NVRAM, related work includes methods for recovering process state from stable storage and for dealing with unreliable memory. Schlichting and Schneider consider the problem of restarting a process that halts due to a processor failure by defining *fault-tolerant actions* that leverage stable storage to persist program state [32]. This work focuses on fault tolerance and considers limited interprocess communication through multi-reader single-writer shared state variables. Several papers consider computation with unreliable memory: Afek et al. consider the consensus problem in this general context [1], Moscibroda and Oshman focus on mutual exclusion [27], and Jayanti et al. propose implementations of shared objects from unreliable base objects [20]. In contrast to these techniques, which break if the number of corruptions exceeds a specified bound, Hoepman et al. [17] as well as Johnen and Higham [21] propose self-stabilizing shared objects that can tolerate any number of memory failures. These objects guarantee wait-free progress, and also ensure that operations return correct values except possibly during the period of instability immediately following a failure.

As regards correctness properties, specifically consistency properties, the dominant ideas in research have long been *serializability* in the context of databases [5], and *linearizability* in the context of in-memory shared objects [16]. Informally speaking, both require that actions – transactions or operations on objects – appear to take effect instantaneously in some serial order. Linearizability further constrains this order so that if operation op_1 *happens before* operation op_2 (i.e., op_1 ends before op_2 begins), then op_1 should precede op_2 in the serial order. *Strict serializability* imposes an analogous constraint on the serial order of transactions. Serializability naturally accommodates crash-recovery failures in the following sense: a transaction interrupted by a failure can simply be aborted and excluded from the serial order. In contrast, linearizability (defined formally in Section 3) has no notion of an aborted or failed operation, and requires that a process finish one operation before it invokes the next. Figure 1 illustrates an execution that is outside the scope of such a model because a process fails in the middle of an operation, then recovers and invokes another operation without completing the previous one.

Frølund et al. address the technicality illustrated in Figure 1 by treating the crash of a process as a response, either successful or unsuccessful, to the interrupted operation [12]. A successful response means that the operation takes effect at some point between its invocation and the crash failure, and an unsuccessful response means that the operation does not take effect at all. This correctness property is stated in [12] as an extension of Lamport's atomicity property for read/write registers [22, 23], and generalized to arbitrary object types by Aguilera and Frølund as *strict linearizability* [2]. The same idea is used by Saito et al. in FAB, a fault-tolerant distributed storage system [31].

Aguilera and Frølund show that strict linearizability has an interesting property in the context of shared memory: it precludes wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers [2]. Intuitively, this is because the effect of a write operation on the implemented object can only be made visible to other processes by a non-atomic series of operations on the single-reader base objects. As a result, when a write is interrupted by a crash, it is sometimes possible for a subsequent read to return either the old or the new value of the implemented object, depending on the identity of the reader. This leads to a scenario where the write appears to take effect after the crash because one read returns the old value and a later read returns the new value.

Guerraoui and Levy propose two correctness properties for read/write registers simulated using message passing in a crash-recovery model [13]. *Persistent atomicity* is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the subsequent invocation of the same process, possibly after the failure. *Transient atomicity* relaxes this criterion even further and allows an interrupted operation to take effect before the subsequent write response of the same process. Although the intent underlying Guerraoui and Levy's definitions was to explore trade-offs between performance and consistency, their properties are quite relevant in the context of shared objects for NVRAM, particularly wait-free implementations that employ *helping mechanisms* whereby an operation invoked by a process p may take effect by the action of another process q after p fails. In contrast, strict linearizability forbids this behavior. Censor-Hillel, Petrank and Timnat recently formalized the concept of helping and showed that without it, certain types of objects lack wait-free linearizable implementations in a conventional shared memory model [6].

Correctness properties for shared objects are easier to reason about when they are *local*, meaning that a collection of objects satisfy a given property P if and only if every object in the collection individually satisfies P [16]. Locality makes it possible to implement and verify shared objects independently, which benefits modularity and concurrency. It is known that linearizability, strict linearizability, and strict serializability are local properties, while ordinary serializability is not. As we explain in Section 4, persistent and transient atomicity are also not local. Vitenberg and Friedman formulate a number of general theorems that can be used to deduce the locality (or lack thereof) of various correctness properties for shared objects [34], however these theorems are proved in a conventional model similar to Herlihy and Wing's, in which a process must complete one operation before it invokes another. Hence, these theorems are not applicable directly in our more general model.

3 Model

Our model is closely based upon Herlihy and Wing's [16].

Processes and objects. We consider N asynchronous *processes*, denoted $p_0, p_1, ..., p_{N-1}$, that communicate by applying operations on shared *base objects* that support atomic operations such as reads, writes, and read-modify-write primitives. Base objects can be used to construct more complex *implemented objects*, such as queues and stacks, by defining *access procedures* that simulate each operation on the implemented object using operations on base objects. Base objects can be *volatile* or *non-volatile*, which determines their behavior during a failure. The state of a process is a collection of *private variables*, including a *program counter*. We consider only one type of failure, called a *system crash* (or crash for short), which resets all volatile base objects as well as the private variables of all processes to their initial values, but preserves the values of all non-volatile base objects. Following a crash a process may either halt permanently or resume its execution (i.e., recover).

Steps and histories. We model the interaction of processes with implemented objects using *steps* and *histories*. There are three types of steps: (1) an invocation step, denoted (INV, p, X, op), represents the invocation by process p of operation op on implemented object X; (2) a response step, denoted (RES, p, X, ret), represents the completion by process p of the last operation it invoked on object X, with response ret; (3) a crash step, denoted (CRASH), denotes a system crash. We include explicit crash steps to accommodate strict linearizability (defined formally in Section 4), but we do not use explicit recovery steps; a process recovers implicitly following a crash by taking an invocation step.

A history H is a sequence of steps, possibly involving multiple processes and implemented objects. For a given history H, we denote by $H|p$ the projection of H onto the steps of process p. Similarly, we denote by $H|O$ the projection of H onto the steps involving implemented object O. We adopt the convention that both $H|p$ and $H|O$ retain all crash steps in H. A response step is *matching* with respect to an invocation step s by a process p on object X in a history H if it is the first response step by p on X that follows s in H, and it occurs before p's next invocation (if any) in H.

Operations. For any history H and any process p, an operation by p in H comprises an invocation step and its matching response, if it exists. An operation is *complete* if it has a matching response step, and *pending* otherwise. Given two operations op_1 and op_2 in a history H, we say that op_1 *happens before* op_2, denoted by $op_1 <_H op_2$, if op_1 has a matching response that precedes the invocation step of op_2 in H. If neither $op_1 <_H op_2$ nor $op_2 <_H op_1$ holds then we say that op_1 and op_2 are *concurrent* in H.

Properties of histories. A history H is *sequential* if no two operations in it are concurrent. Two histories H and H' are *equivalent* if for every process p, $H|p = H'|p$ holds. Every history H must be *well-formed*, meaning that for each process p two conditions hold: (1) each response step in $H|p$ is immediately preceded by an invocation step for which the response is matching, and (2) each invocation step in $H|p$, except possibly the last one, is immediately followed by a matching response or by a crash step. Note that p may have multiple pending operations in $H|p$, in contrast to Herlihy and Wing's model, where at most the last operation may be pending.

Sequential specifications. Every implemented object O has a *sequential specification* that defines its allowed behaviors and is expressed as a set of possible sequential histories over O. A sequential history H is *legal* if for every implemented object O accessed in H, $H|O$ belongs to the sequential specification of O.

4 Correctness Properties

In this section we consider correctness properties for shared objects in NVRAM. Our goal is to identify a small set of candidate properties that could describe the behavior of a variety of shared object implementations. Specifically, we are interested in variations of Herlihy and Wing's linearizability property as it is widely adopted for shared objects in conventional shared memory models. We will give formal definitions of some of the properties discussed in Section 2, discuss their limitations, and then propose an alternative correctness property.

Linearizability itself, as defined by Herlihy and Wing [16], can be formalized in our model but only for histories that are free of crash steps. Given such a history H, we first define its *completion* H' by appending matching responses for a subset of pending operations, and finally removing any remaining pending operations. Appending the responses rather than inserting them between steps of H ensures that H and H' share the same "happens before" relation (i.e., $<_H = <_{H'}$).

▶ **Definition 1** (Linearizability)**.** A finite history H that does not contain any crash steps is linearizable if it has a completion H' and there exists a legal sequential history S such that:
L1. H' is equivalent to S; and
L2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops. appear in S then $op_1 <_S op_2$).

To formalize strict linearizability we introduce a *strict completion* H', which is obtained from H by inserting matching responses for a subset of pending operations after the operation's invocation and before the next crash step (if any), and finally removing any remaining pending operations and crash steps. Inserting responses in this manner may alter the "happens before" relation but guarantees that $<_H \subseteq <_{H'}$ (i.e., if $op_1 <_H op_2$ then $op_1 <_{H'} op_2$).

▶ **Definition 2** (Strict linearizability)**.** A finite history H is strictly linearizable if it has a strict completion H' and there exists a legal sequential history S such that:
SL1. H' is equivalent to S; and
SL2. $<_{H'} \subseteq <_S$ (i.e., if $op_1 <_{H'} op_2$ and both ops. appear in S then $op_1 <_S op_2$).

Referring to H' in clause **SL2** ensures that any operation invoked before a crash in H and completed in H' happens before any operation invoked after the crash. As an example, consider the history H corresponding to Figure 1, where $\perp$ denotes the response of a write:

$$(\text{INV}, p, X, \text{write}(1)), (\text{RES}, p, X, \perp), (\text{INV}, p, X, \text{write}(2)), (\text{CRASH}),$$
$$(\text{INV}, p, X, \text{read}()), (\text{RES}, p, X, 2)$$

A strict completion H' of H can be obtained by inserting a matching response for the write of 2 immediately before the crash step, and then removing the crash step:

$$(\text{INV}, p, X, \text{write}(1)), (\text{RES}, p, X, \perp), (\text{INV}, p, X, \text{write}(2)), (\text{RES}, p, X, \perp),$$
$$(\text{INV}, p, X, \text{read}()), (\text{RES}, p, X, 2)$$

The legal sequential history S that satisfies Definition 2 with respect to H and H' is H' itself, and thus H is strictly linearizable.

Although strict linearizability is an attractive property, Aguilera and Frølund show that it is somewhat restrictive as it forbids wait-free implementations of multi-reader single-writer (MRSW) registers from single-reader single-writer (SRSW) registers [2]. In contrast, linearizability does allow such an implementation, which we discuss further in Section 5.2. Guerraoui and Levy's definitions are less restrictive in comparison as they allow operations interrupted by a failure to take effect after the failure, for example by the action of another process executing a helping mechanism in a wait-free implementation.

Persistent atomicity, which we refer to later on as *persistent linearizability*, can be formalized in our model as follows. Given a history H, a *persistent completion* H' is obtained from H by inserting matching responses for a subset of pending operations after the operation's invocation and before the next invocation step of the same process, and finally removing any remaining pending operations and crash steps.

▶ **Definition 3** (Persistent linearizability). A finite history H is persistently linearizable if it has a persistent completion H' and there exists a legal sequential history S such that conditions **SL1** and **SL2** from Definition 2 hold.

Defining transient atomicity formally in our model presents two technical difficulties, both related to Guerraoui and Levy's definition of a *weak completion* in which a matching response can be added for a pending operation anywhere after the invocation and "before the subsequent *write* reply of the same process." First, it is not obvious how to generalize this concept to arbitrary object types, which may not even support a write operation. Second, the weak completion may contain complete operations invoked by the same process that overlap, even if we restrict our attention to a single object. This not only violates the well-formedness property defined in Section 3 for histories, but also allows operations executed by the same process to take effect in an order different from the order of invocation. (The model in [13] uses a slightly different well-formedness property but the same issues arise.) This anomaly, which we call *program order inversion*, complicates reasoning about the behavior of implemented objects and goes against the intuition underlying Herlihy and Wing's model, in which one process may have at most one pending operation at any given point in time. On the other hand, some reordering among operations may be justifiable in our crash-recovery model for the following reason: if an operation *op* by process p on object X is interrupted by a crash failure, process p should be free to resume execution and invoke an operation on some other object Y independently of any steps on X, by p or any other process, that may cause p's interrupted operation to take effect later on. We refer to this requirement as *independent recovery*, and suggest that it follows naturally from the *nonblocking* property of linearizability: "processes invoking totally-defined operations are never forced to wait" [16].

A more fundamental drawback of both persistent and transient atomicity is the lack of locality, which we consider essential. In both cases the placement of a matching response for a pending operation is constrained by other operations in a manner that may become more restrictive when single-object histories are merged to create a history in which multiple objects are accessed. Figure 2 illustrates a specific example of this problem. Letting H denote the illustrated history, $H|X$ satisfies persistent atomicity because when p's $\text{write}(Y, 1)$ is out of the picture, the remaining operations are permitted to take effect in the following order:

$$p.\text{write}(X, 1),\ q.\text{read}(X) \to 1,\ p.\text{write}(X, 2),\ p.\text{read}(X) \to 2$$

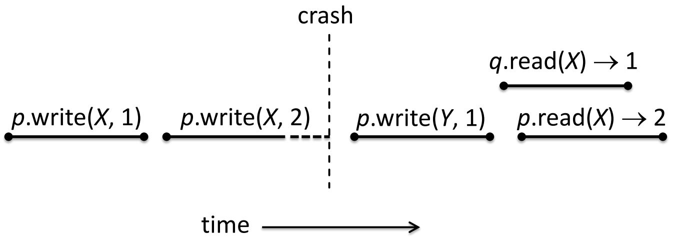


Figure 2 Example execution involving processes p, q and objects X, Y. A failure interrupts p's second write operation before it returns a response.

In particular, p's write$(X, 2)$ is permitted to take effect after q's read(X) as long as it takes effect before the invocation of p's read(X). Similarly $H|Y$ satisfies persistent atomicity because it comprises only a single write operation, namely p's write$(Y, 1)$. However, H itself lacks persistent atomicity because p's write$(X, 2)$ would be forced to take effect before p's write$(Y, 1)$, and hence before q's read(X), which returns 1. An analogous argument shows that transient atomicity is not local because p's write$(X, 2)$ would be forced to take effect before the response of p's write$(Y, 1)$, and hence before q's read(X).

To remedy the technical issues surrounding persistent and transient atomicity, we propose an alternative property called *recoverable linearizability* (or R-linearizability for short) that accommodates arbitrary object types and guarantees locality. Our property is formalized by first defining an appropriate completion procedure, similarly to strict linearizability and persistent atomicity, but deals with the "happens before" relation differently. Given a history H, a *recoverable completion* H' is obtained from H in exactly the same manner as a strict completion. As we discuss shortly, the strictness of the completion does not prevent an operation from taking effect after a failure that interrupts it. However, it does simplify the proof of Theorem 6 later on. Next, we define a precedence order on operations to prevent program order inversion for operations applied to the same object.

▶ **Definition 4.** Given a history H, the *invoked before* relation over pairs of operations in H, denoted $\ll_H$, is an irreflexive partial order defined as follows: if op_1 and op_2 are operations invoked by the same process p on the same object X, and the invocation step of op_1 precedes the invocation step of op_2 in H, then $op_1 \ll_H op_2$.

We use both $<_H$ and $\ll_H$ to constrain the order in which operations appear to take effect:

▶ **Definition 5** (Recoverable linearizability). A finite history H is R-linearizable if it has a recoverable completion H' and there exists a legal sequential history S such that:
RL1. H' is equivalent to S;
RL2. $<_H \subseteq <_S$ (i.e., if $op_1 <_H op_2$ and both ops. appear in S then $op_1 <_S op_2$); and
RL3. $\ll_H \subseteq <_S$ (i.e., if $op_1 \ll_H op_2$ and both ops. appear in S then $op_1 <_S op_2$)

Note that clause **RL2** refers to H and not H', in contrast to clause **SL2** in Definition 2. This ensures that the placement of matching responses in the construction of the recoverable completion, which preserves well-formedness, does not impose undesirable constraints on the order in which operations may appear to take effect. Clause **RL3** compensates for this by disallowing program order inversion at the level of individual objects. Two operations invoked by the same process on different objects may still take effect in an order different from their invocation order, which enables independent recovery. As we show later on in Theorem 7 and Section 5.2, R-linearizability is a local property similarly to strict linearizability, and yet is weak enough to permit a wait-free implementation of MRSW registers from SRSW registers in contrast to strict linearizability.

To illustrate R-linearizability in action, a recoverable completion H' for the history H shown in Figure 2 can be constructed as follows:

$$(\text{INV}, p, X, \text{write}(1)), (\text{RES}, p, X, \bot), (\text{INV}, p, X, \text{write}(2)), (\text{RES}, p, X, \bot),$$
$$(\text{INV}, p, Y, \text{write}(1)), (\text{RES}, p, Y, \bot), (\text{INV}, q, X, \text{read}()), (\text{INV}, p, X, \text{read}()),$$
$$(\text{RES}, q, X, 1), (\text{RES}, p, X, 2)$$

The precedence constraints imposed by clause **RL2** on the legal sequential history S are the transitive closure of the following:

$$p.\text{write}(X, 1) <_S p.\text{write}(X, 2), \quad p.\text{write}(X, 1) <_S p.\text{write}(Y, 1)$$
$$p.\text{write}(Y, 1) <_S q.\text{read}(X), \quad p.\text{write}(Y, 1) <_S p.\text{read}(X)$$

Clause **RL3** imposes the additional constraint $p.\text{write}(X, 2) <_S p.\text{read}(X)$. The legal sequential history S that satisfies Definition 5 with respect to H and H' is the following:

$$(\text{INV}, p, X, \text{write}(1)), (\text{RES}, p, X, \bot), (\text{INV}, p, Y, \text{write}(1)), (\text{RES}, p, Y, \bot),$$
$$(\text{INV}, q, X, \text{read}()), (\text{RES}, q, X, 1), (\text{INV}, p, X, \text{write}(2)), (\text{RES}, p, X, \bot),$$
$$(\text{INV}, p, X, \text{read}()), (\text{RES}, p, X, 2)$$

Thus, the history H shown in Figure 2 is recoverably linearizable or R-linearizable.

In our model it can be shown that strict linearizability is strictly stronger than persistent linearizability, which is strictly stronger than R-linearizability.

▶ **Theorem 6.** *Let H be a history. If H is strictly linearizable (Definition 2) then H is also persistently linearizable (Definition 3), and if H is persistently linearizable then H is also recoverably linearizable (Definition 5). Furthermore, there exists a history that is R-linearizable but not persistently linearizable, and there exists a history that is persistently linearizable but not strictly linearizable.*

Finally, we consider locality in Theorem 7, whose detailed proof is omitted due to lack of space.

▶ **Theorem 7** (locality)**.** *A history H is R-linearizable if and only if, for every object X accessed in H, $H|X$ is R-linearizable.*

Proof Sketch. The "only if" direction follows easily, and so we focus on the "if" direction. Informally speaking, it suffices to define for each object X a linearization point for each operation in $H|X$, such as a base object step at which the operation appears to take effect, and then order the operations in H according to the same linearization points. The main technicality is to show that the linearization points chosen initially are still applicable after the projections $H|X$ are merged together. This point follows easily as long as the definitions of the linearization points are independent of operations on other objects – a property that holds by design in R-linearizability. For a complete operation, clause **RL2** of Definition 5 restricts the linearization point to occur between its own invocation and response, and does not refer to any other operation. For a pending operation, clause **RL3** of Definition 5 restricts the linearization point to occur between its own invocation and the response of the next operation by the same process on the same object (see Definition 4), and also does not refer to operations on any other object. ◀

5 Implementations

In this section we consider techniques for implementing shared objects that are robust against failures in our crash-recovery model in the sense of providing non-blocking progress guarantees in addition to one (or more) of the safety properties formalized in Section 4. A very general but naive technique for constructing such implementations is to take an algorithm designed for the conventional shared memory model, make all program variables non-volatile, and have each process adopt a new and distinct ID on recovery after a crash failure. This approach circumvents enough of the technical issues discussed in Section 4 to make Herlihy and Wing's linearizability property applicable directly, but suffers from two problems. First, it opens the door to program order inversion among operations applied by the same process under different IDs. Second, unless the number of crash failures in a history is bounded, the process IDs grow without bound, leading to a blowup in space complexity for algorithms that store process IDs in variables or use them to index arrays. Merritt and Taubenfeld show that such a blowup is unavoidable for many fundamental problems [24].

Another general strategy for constructing robust objects is to record housekeeping information in NVRAM as processes execute operations on objects, similarly to write-ahead logging in a database, and use it on recovery to repair any operation that was interrupted by a failure. This strategy requires additional writes to NVRAM during failure-free operation, and in practice relies on a specialized recovery procedure that is executed automatically upon recovery while ordinary operations on objects are temporarily suspended (e.g., as in NV-Heaps [8]). Such a recovery procedure is outside the scope of our model, and also implies the use of mutual exclusion to isolate recovery actions from ordinary operations, which is counter to our goal of non-blocking progress.

As an alternative to renaming processes and database-style logging, we explore in this section the following technique: start with a linearizable implementation for ordinary volatile shared memory, make all variables non-volatile, and then modify the algorithm as needed to achieve the desired correctness properties. We focus specifically on two known wait-free implementations: Herlihy's universal construction [14], and a construction of atomic multi-reader single-writer (MRSW) registers from atomic single-reader single-writer (SRSW) registers [15]. In our subsequent discussion of these constructions, we define time complexity as the number of base object operations executed per implemented operation between its invocation and either its response or a failure, whichever occurs first.

5.1 Herlihy's Wait-Free Universal Construction

Herlihy [14] proposes a construction of wait-free linearizable objects that is *universal* – it can implement any shared object type. The construction, which we reproduce in Figure 3, works as follows. Each process, when it wants to apply an operation on the implemented object, attempts to have a *cell* structure representing its invocation threaded onto a linked list of such structures. This list determines both the subset of operations that have taken effect and their respective linearization order. The invocation is passed to the access procedure Universal as a structure of type *INVOC*, which encodes the operation to be applied and its arguments. At line 2, the process announces its invocation to others by storing a pointer to the corresponding cell at a dedicated element of the array *Announce*. Each process also has a dedicated element in the array *Head*, and uses a scan of this array at lines 3–5 to identify a cell near the end of the linked list. The *max* operator at line 4 compares cell structures by their *seq* member, which is a sequence number indicating the position of a cell in the linked list. Arrays *Announce* and *Head* are initialized with all elements pointing to a special anchor cell, which represents the start of the list and has a sequence number of one.

Base objects:
Announce, *head*: array[0..N-1, 0..N-1] of
pointer to cell, each element initialized to the address of the anchor cell

```
Function Universal(what: INVOC) for process p
1  mine: cell := [seq: 0, inv: what,
     new: consensus object, before: NULL,
     after: consensus object]
2  Announce[p] := mine
3  foreach process ID q do
4      Head[p] := max(Head[p], Head[q])
5  end
6  while Announce[p].seq = 0 do
7      c: pointer to cell := Head[p]
8      help: pointer to cell :=
         Announce[(c.seq mod N) + 1]
9      if help.seq = 0 then
10         prefer := help
11     else
12         prefer := Announce[p]
13     end
14     d := decide(c.after, prefer)
15     decide(d.new, apply (d.inv, c.new.state))
16     d.before := c
17     d.seq := c.seq + 1
18     Head[p] := d
19 end
20 Head[p] := Announce[p]
21 return Announce[p].new.result
```

Figure 3 Herlihy's universal wait-free construction for N processes [14].

The universal construction deals with concurrency using two consensus objects per cell: *after* is a pointer to the next cell in the list and deals with concurrent attempts to thread a cell onto the list; *new* is a structure that holds the state of the implemented object (in *new.state*) and the corresponding response (in *new.result*), and deals with concurrent attempts to determine the state transition for an invocation when the transition function denoted by "apply" at line 15 is nondeterministic.

For wait-freedom the construction uses a helping mechanism whereby a process attempting to thread a new cell onto the linked list at line 14 may act on a cell announced by another process, which is chosen at lines 8–13. This mechanism ensures that every cell that is announced is threaded onto the linked list in a bounded number of base object operations, as long as some process continues to take steps.

Let *UC* denote Herlihy's universal construction in our model with all base objects made non-volatile, and let *UC′* denote the same implementation but with the *Announce* array made volatile. In the remainder of this section we show that *UC* provides R-linearizability but not persistent or strict linearizability, and that *UC′* provides all three properties. Detailed proofs of correctness for Theorems 8–10 are omitted due to lack of space.

▶ **Theorem 8.** *Every finite history H of implementation UC is R-linearizable.*

Proof sketch. Intuitively, we must show that when an operation applied by a process p is interrupted by a failure, it is safe for p to abandon this operation and start executing procedure Universal from line 1 when it invokes its next operation. To that end, we prove that p's abandoned operation takes effect at most once, and moreover it never takes effect out of order with respect to any operation that p invokes after the failure on the same instance of the universal construction. The key to the proof is the observation that each time p executes procedure Universal, it overwrites its element of the *Announce* array at line 2, which has two implications. First, the cell associated with p's interrupted operation becomes inaccessible to future iterations of the helping mechanism, unless that cell has already been threaded onto the linked list. Second, any iteration of the helping mechanism that is acting on that cell and has already begun prior to p's execution of line 2, is doomed to fail if any other cell is threaded onto the linked list first. The detailed proof uses both points to establish R-linearizability for all histories of *UC*. ◀

▶ **Theorem 9.** *Implementation UC for $N \geq 2$ processes has a finite history H that is neither strictly or persistently linearizable.*

Proof sketch. We show that when an operation applied by a process p is interrupted by a failure, this operation may take effect after p's next invocation step by the action of another process that is participating in the helping mechanism. ◀

▶ **Theorem 10.** *Every finite history H of implementation UC' is strictly linearizable.*

Proof sketch. Extending the proof of Theorem 8, we show that when an operation applied by a process p is interrupted by a failure, its cell is either threaded before the failure or not at all, since the array *Announce* is volatile in UC'. ◀

▶ **Theorem 11.** *Implementations UC and UC' for N processes have time complexity $O(N)$.*

Proof. As in the original analysis [14] it can be shown that the while loop has at most $N + 1$ iterations during any execution of the procedure Universal. Furthermore, the loop at lines 3–5 has exactly N iterations. Since each loop iteration applies $O(1)$ base object operations, this implies the claimed time complexity. ◀

Our discussion of *UC* and UC', which are extensions of Herlihy's universal construction, shows that any object type can be implemented in a wait-free and R-linearizable manner in our crash-recovery model using base objects of types read/write register and consensus. Thus, consensus is universal in our model just as in Herlihy's [14]. Furthermore, UC' demonstrates that it is possible to achieve wait-freedom and strict linearizability (hence freedom from program order inversion) simultaneously in a construction that depends crucially on a helping mechanism. Specifically, the use of volatile base objects as elements of the array *Announce* is sufficient to ensure a "clean shutdown" of the helping mechanism during a failure.

5.2 Implementation of MRSW Registers from SRSW Registers

As our second example we analyze a wait-free implementation of atomic multi-reader single-writer (MRSW) registers from atomic single-reader single-writer (SRSW) registers. The construction, which we refer to as MRSW, is presented in Section 4.2.5 of [15] and is similar to Israeli and Li's [19]. We chose this example because it illustrates the case when a known implementation fails to satisfy R-linearizability "out of the box," even if we assume that

Base objects:

A: shared array[0..N-1, 0..N-1] of record [V: value, T: timestamp] initialized to $(V_0, 0)$ where V_0 is the implemented type's initial value

$T_{\max}$: integer, initially 0

```
Function write(V: value) for process p_w
$T := T_{\max} + 1$
$T_{\max} := T$
foreach $i$: int in 0..$N$-1 do
    $A[i, i] := (V, T)$
end
```

```
Function read() for process p_i
$(V_i, T_i) := (\text{NULL}, -1)$
foreach $j$: int in 0..$N$-1 do
    $(V_{temp}, T_{temp}) := A[j, i]$
    if $T_{temp} > T_i$ then
        $(V_i, T_i) := (V_{temp}, T_{temp})$
    end
end
foreach $j$: int in 0..$N$-1 do
    if $j \neq i$ then
        $A[i, j] := (V_{temp}, T_{temp})$
    end
end
return $V_{temp}$
```

Figure 4 Implementation of atomic MRSW registers from atomic SRSW registers [15].

base objects are non-volatile. Furthermore, as we explain later on, a modified R-linearizable version of this implementation separates R-linearizability from strict linearizability. For completeness the pseudo-code for the implementation is included in Figure 4.

The implemented object is represented using an array $A[0..N\text{-}1, 0..N\text{-}1]$ of SRSW register base objects. The distinguished writer process, denoted p_w for some $w \in 0..N$-1, maintains a variable $T_{\max}$, initially 0, for timestamping write operations. Each process p_i is able to read column i, and write row i with the exception of the diagonal element $A[i, i]$, which is only written by p_w. Each element of A is of the form (V, T), where V is a value written to the implemented MRSW register and T is a timestamp assigned by the writer. In the initial state, V holds the implemented register's initial value and $T = 0$ for each array element. To apply a write(V) operation, process p_w increments $T_{\max}$ to obtain a timestamp T higher than any prior timestamp, and writes (V, T) to the diagonal elements of A. To apply a read operation, process p_i reads the highest timestamp T_i in column i to determine the corresponding latest value V, then writes (V, T_i) to each element in row i except $A[i, i]$, and finally returns V. By writing (V, T_i) in row i, p_i announces its response to other readers, which ensures that subsequent reads do not return older values. (Values are ordered naturally in terms of "age" because there is only one writer.) This register implementation has time complexity $O(N)$ for N processes, and is linearizable in the absence of failures.

Before analyzing the implementation in the crash-recovery model, a subtle detail must be settled: we assume that by default p_w writes the diagonal elements in the order specified by the pseudo-code, namely from $A[0, 0]$ to $A[N\text{-}1, N\text{-}1]$. This point is irrelevant in a conventional asynchronous model with permanent crash failures, but as we explain shortly, it is crucial for correctness in our crash-recovery model. Assuming that both A and $T_{\max}$ are non-volatile base objects, which is necessary for $T_{\max}$ to increase monotonically whenever the implemented register object is written, the MRSW register construction violates R-linearizability in our crash recovery model. Thus, the construction does not work correctly "out of the box," in contrast to Herlihy's construction. This result is stated in Theorem 12, whose detailed proof is omitted due to lack of space.

▶ **Theorem 12.** *For any number of processes $N \geq 2$, implementation MRSW has a history that is not R-linearizable.*

```
Function write(V: value) for process $p_w$
$T := T_{\max} + 1$
$T_{\max} := T$
$A[w, w] := (V, T)$
foreach $i$: int in 0..$N$-1 do
    if $i \neq w$ then
        $A[i, i] := (V, T)$
    end
end
```

Figure 5 Access procedure for the write operation of implementation MRSW′.

Proof sketch. Consider $N = 2$, with p_1 being the designated writer. The implementation has a history where p_1 begins a write(1) operation and a crash failure occurs after p_1 has written only one of the diagonal elements of A, namely $A[0, 0]$. Next, p_1 executes a read() and obtains the initial value, say 0, from $A[0, 1]$ and $A[1, 1]$. Finally, p_0 executes a read() and obtains the new value, namely 1, from $A[0, 0]$. Thus, p_1's interrupted write appears to take effect not only after the failure but also after its subsequent read() operation.[1] ◀

▶ **Corollary 13.** *For any number of processes $N \geq 2$, implementation MRSW has a history that is not persistently linearizable or strictly linearizable.*

The proof of Theorem 12 not only illustrates a weakness of implementation MRSW with respect to R-linearizability, but it also suggests a remedy. That is, while executing a write operation, the distinguished writer process p_w should overwrite the diagonal elements of array A starting with the row and column corresponding to its own process ID. We refer to the modified implementation as MRSW′, and present the pseudo-code for the modified write access procedure in Figure 5. The correctness of MRSW′ is established in Theorems 14–16. The detailed proofs are omitted due to lack of space.

▶ **Theorem 14.** *Every finite history H of implementation MRSW′ is R-linearizable.*

Proof sketch. Writing the diagonal elements of A in the modified order addresses the specific problem described in the proof sketch of Theorem 12 because p_1's read() operation correctly observes the value assigned by p_1's earlier write(1). Thus, the interrupted write appears to take effect before the writer's subsequent operation on the same object. The case of an interrupted write followed by another write is also dealt with correctly, as the latter operation overwrites all the base objects accessed by the former. ◀

▶ **Theorem 15.** *For any number of processes $N \geq 2$, implementation MRSW′ has a finite history that is neither persistently nor strictly linearizable.*

Proof sketch. Consider $N = 2$, with p_1 being the designated writer. The implementation has a history where p_1 begins a write(1) operation and a crash failure occurs after p_1 has written only one of the diagonal elements of A, namely $A[1, 1]$. Next, p_1 invokes a read() operation

[1] In a conventional crash failure model, there is no need to implement a read() operation for the designated writer because this process can record the last value written to the implemented register using a private variable. In contrast, in our model the value of such a private variable would be lost during a failure.

but does not yet access any base objects. Process p_0 then races ahead and completes a read() that obtains the initial value, say 0, from $A[0,0]$ and $A[1,0]$. Finally, p_1 completes its read() and obtains the new value, namely 1, from $A[1,1]$. In this history, p_1's interrupted write(1) operation appears to take effect after p_0's read(), hence after the invocation step of p_1's read(), which violates both persistent and strict linearizability. ◀

▶ **Theorem 16.** *Implementations MRSW and MRSW′ for N processes have time complexity* $O(N)$.

Theorems 14–15 separate R-linearizability from strict linearizability in the following sense: whereas a strictly linearizable wait-free implementation of MRSW registers from atomic SRSW registers is impossible in an asynchronous model with crash failures, an R-linearizable wait-free implementation is possible in our asynchronous model with crash-recovery failures.

6 Conclusion

In this paper we defined a shared memory model with crash-recovery failures and a combination of volatile and non-volatile main memory. We then surveyed a number of safety properties inspired by linearizability that address the behavior of operations interrupted by failures in our model, identified the limitations of these properties, and proposed an alternative property called R-linearizability. Finally, we discussed implementation techniques.

Our coverage of implementation techniques centers around an approach where a known linearizable implementation designed for a conventional shared memory model is instantiated by making all base objects non-volatile, and then transformed as needed to yield R-linearizability. We showed that Herlihy's construction is R-linearizable "out of the box", and can be made strictly linearizable using an additional transformation that prevents the helping mechanism from acting on operations invoked prior to the most recent failure. In contrast, we showed that the MRSW register construction is not immediately R-linearizable, but can be made R-linearizable using a transformation that changes the order in which base objects are written. As shown by Aguilera and Frølund, a wait-free strictly linearizable MRSW register implementation from SRSW registers is impossible [2], and thus our transformed MRSW construction separates R-linearizability formally from strict linearizability.

Acknowledgments. Ryan Berryhill contributed to this research while enrolled in undergraduate studies at the University of Waterloo, ECE Department. Wojciech Golab and Mahesh Tripunitara are supported in part by the NSERC Discovery Grants Program. The authors are grateful to the anonymous referees and to Vassos Hadzilacos for their feedback on this work.

References

1 Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *Proc. 11th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 47–58, 1992.

2 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, Palo Alto, CA, USA, November 2003.

3 Marcos K. Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

4 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

5 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

6 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proc. of the 34th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, 2015.

7 Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

8 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

9 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

10 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, August 1965.

11 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.

12 Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. Building storage registers from crash-recovery processes. Technical Report HPL-SSP-2003-14, HP Labs, Palo Alto, CA, USA, 2003.

13 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004.

14 Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.

15 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

16 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, July 1990.

17 Jaap-Henk Hoepman, Marina Papatriantafilou, and Philippas Tsigas. Self-stabilization of wait-free shared memory objects. *J. Parallel Distrib. Comput.*, 62(5):818–842, 2002.

18 Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *Proc. of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–14, 2010.

19 Amos Israeli and Ming Li. Bounded time-stamps. In *Proc. of the 28th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 371–382, 1987.

20 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45:451–500, 1998.

21 Colette Johnen and Lisa Higham. Fault-tolerant implementations of regular registers by safe registers with applications to networks. In *Proc. of 10th International Conference of Distributed Computing and Networking (ICDCN)*, pages 337–348, 2009.

22 Leslie Lamport. On interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

23 Leslie Lamport. On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

24 Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proc. of the 14th International Conference on Distributed Computing (DISC)*, pages 164–178, 2000.

25 Jeffrey C. Mogul, Eduardo Argollo, Mehul A. Shah, and Paolo Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

26 C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

27 Thomas Moscibroda and Rotem Oshman. Resilience of mutual exclusion algorithms to transient memory faults. In *Proc. of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 69–78, 2011.

28 Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 225–238, 2013.

29 David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, second edition, 1997.

30 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proc. of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 265–276, 2014.

31 Yasushi Saito, Svend Frølund, Alistair C. Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58, 2004.

32 Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

33 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

34 Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *Proc. of the 17th International Symposium on Distributed Computing (DISC)*, pages 92–105, 2003.

35 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011.

The Benefits of Entropy in Population Protocols

Joffroy Beauquier[1], Peva Blanchard[*2], Janna Burman[†3], and Rachid Guerraoui[4]

1 **LRI, Paris-South University, Orsay, France**
joffroy.beauquier@lri.fr
2 **LPD, EPFL, Lausanne, Switzerland**
peva.blanchard@epfl.ch
3 **LRI, Paris-South University, Orsay, France**
janna.burman@lri.fr
4 **LPD, EPFL, Lausanne, Switzerland**
rachid.guerraoui@epfl.ch

Abstract

A distributed computing system can be viewed as the result of the interplay between a distributed algorithm specifying the effects of a local event (e.g. reception of a message), and an *adversary* choosing the interleaving (schedule) of these events in the execution. In the context of large networks of mobile pairwise interacting agents (population protocols), the adversary models the mobility of the agents by choosing the successive pairs of interacting agents. For some problems, assuming that the adversary selects the schedule according to some probability distribution greatly helps to devise (almost) correct solutions. But how much randomness is really necessary? To what extent does a problem admit implementations that are robust against a “not so random” schedule?

This paper takes a first step in addressing this question by borrowing the concept of T-*randomness*, $0 \leq T \leq 1$, from algorithmic information theory. Roughly speaking, the value T fixes the *entropy rate* of the considered schedules. For instance, the case $T = 1$ corresponds, in a specific sense, to schedules in which the pairs of interacting agents are chosen independently and uniformly (perfect randomness). The holy grail question can then be precisely stated as determining the *optimal entropy rate* to solve a given problem.

We first show that perfect randomness is never required. Precisely, if a finite-state algorithm solves a problem with 1-randomness, then this algorithm still solves the same problem with T-randomness for some $T < 1$. Second, we illustrate how to compute bounds on the optimal entropy rate of a specific problem, namely the *leader election* problem.

1998 ACM Subject Classification F. Theory of Computation, F.1.0 General, F.1.2 [Modes of Computation] Parallelism and Concurrency

Keywords and phrases algorithmic randomness, entropy, leader election, distributed computing, scheduler, population protocols

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.21

1 Introduction

The way that events in a distributed system are triggered depends on some external causes, often referred to as the environment. To model the environment, an abstraction, called

[*] Acknowledgment: the European ERC Grant 339539 – AOC.

[†] Acknowledgment: the Israeli-French Maimonide and the INS2I PEPS JCJC research grants.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 21; pp. 21:1–21:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

scheduler, is introduced. The scheduler specifies which sequences of events are possible and which ones are impossible. As the correctness of a distributed algorithm depends both on the algorithm and on the scheduler, this latter is often considered as an adversary. In this context, one can think of the scheduler as trying to trigger a sequence of events that will fool the algorithm. Most of the impossibility proofs rely on exhibiting a particular schedule for which the specification of the problem is not satisfied. One way to circumvent such impossibility proofs is to assume that the adversary selects a *random* schedule. This assumption is generally not in contradiction with real environments, which endure phenomena like variations of temperature, power supply, network traffic, etc., in sort of a randomized way.

However, it is not as obvious as it may seem at first sight that a given environment yields truly random schedules. Actually, truly random sequences are very hard to get. For instance, the scheduling of processes in a multi-core architecture depends on a physical process which may exhibit a partially predictable behaviour. Or, this scheduler may rely on some algorithm using a pseudo-random source, which is not truly random. In other settings, like mobile networks, the interactions between nodes may follow some regularity because, e.g., the mobility area is limited, or some paths are statistically preferred to others, etc.

This raises the following interesting question: to what extent an algorithm may exhibit some robustness against *imperfect* randomness? And, for a specific problem, what is the optimal robustness that one may hope to achieve? The main goal of this paper is to tackle these important questions.

A first step towards this goal is to lay down a definition of randomness, and a measure of robustness, which are amenable to analysis. Randomness and probability theory are obviously strongly related, but here probability theory does not help because it does not allow to qualify an individual schedule as being random. In this context, probability theory is more about measuring the number of, say, "bad" schedules, which leads to notions of solving a problem almost surely, or with high probability.

Algorithmic information theory, on the other hand, allows to define what it means for an individual schedule to be random. Intuitively, the complexity of a finite schedule is defined as the length of the shortest computer program able to produce this schedule; and the schedule is random if no program is substantially shorter than the sequence itself[1]. This intuition extends to infinite schedule by considering how the complexity of its prefixes grow. More precisely, we borrow the concept of T-randomness, $0 \leq T \leq 1$, from [23, 24], where T measures the *entropy rate* of a schedule, i.e., the complexity growth of the schedule's prefixes. Roughly speaking, the larger is T, the more random is the schedule. In particular, the case of $T = 1$ represents perfect randomness.

This notion allows to precisely quantify the robustness of an algorithm against imperfect randomness: this is simply the least entropy rate T such that any T-random schedule induces an execution that still satisfies the problem's specification. In this context, a natural issue is to determine, given a problem P, the optimal entropy rate T such that some algorithm solves P for all T-random schedules.

We illustrate this notion in a simple distributed computing model (population protocols [5]). Our contributions are twofold. First, we show that perfect randomness is never required in finite-state systems (Section 4). That is, whenever a finite-state algorithm solves a problem P over the 1-random schedules, then this algorithm solves the same problem P over the T-random schedules for some $T < 1$. That is, the optimal entropy rate of P is strictly less

[1] Roughly speaking, the shortest computer program just enumerates the successive events of the schedule.

than 1. Moreover, our proof exhibits a general method to compute upper bounds on the optimal entropy rate.

Our second contribution focuses on a specific problem, fundamental in distributed computing, namely *leader election* (Section 5). In this problem, all processes start in the same initial state (with the same initial knowledge), and a unique process (the elected leader) eventually permanently outputs 1 while the others output 0. The method exhibited in the previous part (Section 4) is applied to derive an explicit upper bound on the optimal entropy rate of leader election. Next, we compute a lower bound T_{sym}. This bound exploits the relation between leader election and symmetry breaking. Indeed, some schedulers are able to produce schedules which "maintain symmetry", in the sense that any process have the same state as some other process; thereby preventing the election of a unique leader. The bound T_{sym} quantifies exactly the maximum entropy rate above which a schedule cannot be symmetric in the previous sense.

The rest of the paper is organized as follows. Section 2 recalls basic definitions of algorithmic information theory, as well as the distributed computing model we consider. In Section 3, we introduce the notions of T-random adversary and optimal entropy rate of a distributed computing problem. Our first contribution, i.e., proving that perfect randomness is never required, is presented in Section 4. We give upper and lower bounds for the entropy rate of leader election in Section 5.

For the sake of clarity, some proofs are postponed to the appendix.

Related Work. Algorithmic information theory has started with the seminal work of Solomonoff [20, 21] and Kolmogorov [16]. One of the major achievements of this field was a precise definition of randomness. In [18], Martin-Löf defines random sequences as those which withstand all effective statistical tests. In [10], an equivalent formulation of a random sequence is given, as one whose shortest (prefix-free) program has the same length as the schedule. In [23, 24], Tadaki generalizes this formulation, and introduces the notion of partial randomness, namely T-randomness ($0 \leq T \leq 1$). The original Martin-Löf's definition of randomness then coincides with 1-randomness.

The computational model used in this paper, known as population protocol, has been introduced by Angluin *et al.* in [5] to model large wireless networks of anonymous finite-state mobile processes interacting pairwise. In much of the literature on this model [5, 6, 4], the scheduler is subject to a fairness condition, namely global fairness, which depends on the algorithm being run: if a configuration is reachable infinitely often, then this configuration is reached infinitely often. Actually, the proof in Section 4 (indirectly) shows that 1-randomness, which is a condition independent of the algorithm being run, implies global fairness for any finite-state algorithm. In particular, any algorithm working under global fairness is guaranteed to work under 1-randomness.

Being an important primitive in distributed computing, leader election has been extensively studied. In particular, in [3], Angluin relates the notion of graph coverings with the impossibility of leader election, and more generally, to any problem that requires "symmetry breaking". This approach has been proved to be fruitful in other contexts as well [12, 19]. To circumvent this issue, many approaches [15, 1, 13] have proposed randomized algorithms, each process having access to some random sequence. These algorithms are probabilistic and solve the problem at hand, either almost surely, or with high probability, but not exactly. Our approach is orthogonal. We consider deterministic algorithms, the randomness being put entirely on the side of the adversary. In some sense, our approach is closer to the works [2, 7], where random scheduling helps solving the problem at hand; except that our motivation is to assess the amount of randomness required. In [8], thanks to the almost random nature of global fairness, authors propose a deterministic algorithm solving leader election over arbitrary graphs.

2 Background Definitions

Let X be a finite set, $|X|$ its cardinality, and X^* (resp. X^ω) the set of finite (resp. infinite) sequences on X. The length of a finite sequence $w \in X^*$ is denoted by $|w|$. For any $w \in X^*$, we denote by $w \upharpoonright n$ the prefix of w of length n. The concatenation of two sequences $u, v \in X^*$ is denoted by uv. We denote by u^k the concatenation of k copies of u.

2.1 Algorithmic Information Theory

Here, we briefly recall the basic definitions and properties. For further details, please refer to, e.g., [17, 9, 22, 25]. Our presentation is slightly different from the usual one as we will be dealing with several alphabets.

Consider finite alphabets X, Y. A *(partial) recursive function* $M : X^* \to Y^*$ is a function computed by some Turing machine (with input alphabet X, and output alphabet Y). The function M is *prefix-free* if its domain does not contain two elements, one of which being a prefix of the other. It is known that there exists a *universal prefix-free recursive function* $U_{XY} : X^* \to Y^*$. Intuitively, this function is universal in the sense that it can simulate any other prefix-free recursive function (of type $X^* \to Y^*$).[2] We fix such a universal function U_{XY}.

The following defines the complexity of a finite word $S \in X^*$ as the length (in bits) of the smallest word p (with the same alphabet) such that the universal function $U_{XX}(p)$ produces S. Intuitively, the word p is the *program* which computes S when executed on the machine U.

▶ **Definition 1** (Complexity)**.** The complexity of a sequence $S \in X^*$ is $H_X(S) = |p| \log |X|$ where $p \in X^*$ is the shortest sequence such that $U_{XX}(p) = S$.

The factor $\log |X|$ is simply a rescaling factor, so that the complexity is expressed in bits.

We now consider infinite sequences, i.e., elements of X^ω. The notion of partial randomness from [23] allows to quantify the degree of randomness of an infinite sequence by looking at how the complexity of its prefixes grows. We adapt Tadaki's definition to the case of an arbitrary alphabet.

▶ **Definition 2** (T-randomness)**.** Given a real value $T \in [0, 1]$, $S \in X^\omega$ is *T-random on X* if, for all n, $H_X(S \upharpoonright n) \geq T \cdot n \cdot \log |X| - O(1)$.

All the entropy-related computations in the paper rely solely on the following lemmas. These are adaptations of known results of algorithmic information theory [17, 9]. For the reader's convenience, we give an intuitive interpretation of Lemma 3 in the appendix (Section A).

▶ **Lemma 3.**

(a) *Let $q : X^* \to Y^*$ be a partial recursive function. Then, for all $S \in X^*$, $H_Y(q(S)) \leq H_X(S) + O(1)$.*

[2] Formally, there exists an effective enumeration $(M_i)_{i \in X^*}$ of all the prefix-free recursive function (type $X^* \to Y^*$) such that, for all $i, p \in X^*$, $U_{XY}\langle i, p\rangle = M_i(p)$.

(b) *For all $S \in X^*$, $H_X(S) \leq |S| \cdot \log |X| + 2 \cdot \log |S| + O(1)$.*

(c) *Let $A \subseteq X^* \times \mathbb{N}$ be a recursively enumerable set such that the subset $A_n = \{w : (w, n) \in A\}$ is finite for every $n \in \mathbb{N}$. Then, for all n, for every $S \in A_n$ with $|S| = n$, $H_X(S) \leq \log |A_n| + 4 \cdot \log n + O(1)$.*

(d) *An infinite sequence $S \in X^\omega$ is T-random if and only if any of its suffixes is T-random.*

▶ **Lemma 4** (Existence of a T-random sequence)**.** *For any $0 \leq T \leq 1$, there exists an infinite sequence $S \in X^\omega$ which is T-random on X.*

2.2 Computational Model

The computational model used in this paper was introduced in [5]. The model involves two parts: a graph representing the possibilities of interactions between the processes, and a list of rules (the algorithm) describing how the states of two interacting processes are updated.

Formally, a *communication graph* is a directed graph G (without self-loops). Each node x represents a *process*. Each directed edge (x, y) represents a possible meeting event in which x is the initiator and y is the responder. We denote by $\mathcal{V}(G)$ and $\mathcal{E}(G)$ the set of processes (vertices) and meeting events (edges) of G respectively. In this work, unless stated otherwise, every graph is assumed to be weakly connected. A *schedule* on G is a sequence of edges of G, that is, a finite or infinite sequence on the alphabet $\mathcal{E}(G)$. An *assignment on* G is a map that associates with every vertex in G some value (in some given set). A *trace on* G is a sequence of assignments on G.

An *algorithm* $\mathcal{A}$ is a tuple $(Q, X \xrightarrow{I} Q, Q \xrightarrow{O} Y, Q^2 \xrightarrow{\delta} Q^2)$ where Q is a set called the state space, $X \xrightarrow{I} Q$ is the *input function*, $Q \xrightarrow{O} Y$ is the *output function*, and $Q^2 \xrightarrow{\delta} Q^2$ is the *transition function*. Note that we consider only *deterministic* algorithms (the transition function is single valued).

An *input assignment* (resp. *output assignment*) is an assignment with values in X (resp. Y). A *configuration* is an assignment with values in the state space Q. Each input assignment α yields an *initial configuration* $I \circ \alpha$. Similarly, each configuration γ yields an output assignment $O \circ \gamma$.

Given an edge $e = (x, y)$ in G and two configurations γ, γ', we write $\gamma \xrightarrow{e} \gamma'$ when $(\gamma'(x), \gamma'(y)) = \delta(\gamma(x), \gamma(y))$ and, for all $z \notin \{x, y\}$, $\gamma(z) = \gamma'(z)$. An (finite or infinite) execution of $\mathcal{A}$ on G is a sequence $\gamma_0 \xrightarrow{e_1} \gamma_1 \dots$ where γ_0 is the initial configuration. The sequence of edges of G appearing in the execution is the underlying schedule of the execution. Note that an execution is entirely determined by its underlying schedule and the initial configuration. It is assumed that every edge occurs infinitely often in the schedule. The *output trace* of an execution $\gamma_0 \xrightarrow{e_1} \gamma_1 \dots$ is the corresponding sequence $(O \circ \gamma_0)(O \circ \gamma_1) \dots$ of output assignments.

3 Entropy of Schedules

Since schedules are infinite sequences on the alphabet $\mathcal{E}(G)$, we can apply the concept of T-randomness to classify them. This motivates the following definition.

▶ **Definition 5** (Adversary $\mathbb{A}(T, G)$)**.** Given a real value $T \in [0, 1]$, and a communication graph G, the *T-random adversary* $\mathbb{A}(T, G)$ is the set of infinite schedules $S \in \mathcal{E}(G)^\omega$ such that S is T-random on $\mathcal{E}(G)$.

For the sake of simplicity, schedules of $\mathbb{A}(T, G)$ are simply said to be T-random on G. We denote by $H_G(\cdot)$ the quantity $H_{\mathcal{E}(G)}(\cdot)$.

A *decision problem* on G is a tuple (P, X, Y) where X is the input alphabet, Y the output alphabet, and P is a function that associates with every input assignment α (with values in X) on G, a set $P(\alpha)$ of output assignments (with values in Y) on G. The input and output alphabets are often implicitly assumed, and we refer to P as the problem directly.

An algorithm $\mathcal{A}$ is said to *solve a problem P on G under adversary $\mathbb{A}(T, G)$* if, for every schedule $S \in \mathbb{A}(T, G)$, for every input assignment α, the execution induced by S and α yields an output trace having a suffix $\beta\beta\ldots$ where $\beta \in P(\alpha)$. Intuitively, this means that the output of the algorithm eventually stabilizes to a legal output assignment, given the input assignment the execution started with.

▶ **Definition 6** (Optimal entropy rate). The *optimal entropy rate of P on G* is $T(P, G) = \inf\{T : \text{some algorithm solves } P \text{ on } G \text{ with adversary } \mathbb{A}(T, G)\}$. If the problem is impossible to solve with any $\mathbb{A}(T, G)$, we set $T(P, G) = \infty$.

▶ Remark. Note that we classify the schedules according to their entropy rate only. Hence, this classification applies to a very broad spectrum of adversary schedulers, from, e.g., typical schedules of a uniform bernoullian scheduler, to, e.g., those of a possibly non-markovian one. If an algorithm is proven to solve a problem for all T-random schedules, then it does not matter how exactly the real schedules are produced: as long as they are all T-random, the algorithm will work.

4 Perfect Randomness is Never Required

▶ **Proposition 7.** *Let P be a problem, and G a graph. If there exists a protocol $\mathcal{A}$ with finite state space solving problem P under adversary $\mathbb{A}(1, G)$, then there exists $0 \leq T < 1$ such that $\mathcal{A}$ solves P under $\mathbb{A}(T, G)$. In particular, the optimal entropy rate of P on G is strictly less than* 1.

The main idea consists in the analysis of the *transition graph* of the protocol, whose nodes represent the configurations, and the (directed) edges, the transitions between configurations. The transition graph can be partitioned into strongly connected components. The final components, i.e., the components with no out-going edges play a particular role. Indeed, a 1-random schedule necessarily drives the system towards a final component, and makes it visit every configuration in that component infinitely often. Thus, by the assumption on the protocol, the configurations in this component produce the same output assignment satisfying the problem's specification. In particular, it suffices to drive the system into one final component to yield an execution satisfying the problem's specification.

The proof below relies on the observation that, if an execution is stuck into a non-final component C, then its underlying schedule repeatedly avoids some pattern of events which would drive the system out of this component. This repeated dodge imposes an upper bound $t(C) < 1$ on the underlying schedule. Taking the contrapositive, if the schedule were T-random with $t(C) < T < 1$, then the corresponding execution would escape the component C. Since the protocol is finite-state, there are finitely many non-final components, and it suffices to take $\max_C t(C) < T < 1$ for any T-random schedule to escape any non-final component. We now give the detailed proof.

Proof.

Basics. We define the transition graph $\Gamma = \Gamma(\mathcal{A})$ as the edge-labeled directed graph whose nodes are the configurations reachable from the initial configurations, and the edges denote the transition $\gamma \xrightarrow{e} \gamma'$ where e is an event (an edge of G). The underlying schedule of a finite path in Γ is the sequence of successive labels (edges of G) along the path. Since $\mathcal{A}$ has a finite state space, the graph Γ has finitely many nodes.

We can decompose Γ in strongly connected components. A *final* component is a component without any out-going transitions. Given a component C in Γ, we define for each natural number n, the set $C[n]$ of underlying schedules of paths of length n in C. Then, letting d be the number of edges in G, we define $t(C) = \liminf_{n\to\infty} \frac{\log|C[n]|}{n\log d}$ and $T_0 = \max t(C)$ over all the non-final components C.

$T_0 < 1$. We prove that $T_0 < 1$. Consider any non-final component C. We build a finite schedule u such that applying u to any configuration in C leads outside of C. Because it is non-final, there exists configuration $\gamma_0 \in C$, and an event u_0 such that the transition $\gamma_0 \xrightarrow{u_0} \gamma_0' \notin C$. We consider the set D_0 of configurations such that applying u_0 to any of them leads outside of C. If D_0 comprises all the configurations of C, then we are done. Otherwise, pick any configuration $\gamma_1 \in C - D_0$; then $\gamma_1 \xrightarrow{u_0} \gamma_1' \in C$. Since C is strongly connected, there exists a finite schedule w_1 such that applying w_1 to γ_1' leads to some configuration in D_0. Therefore, applying the schedule $u_1 = u_0w_1u_0$ to any configuration in $D_0 \cup \{\gamma_1\}$ leads outside of C. We can consider the set D_1 of configurations in C such that applying the schedule u_1 to any of them leads outside of C. If D_1 comprises all the configurations of C, then we are done. Otherwise, we can repeat the same procedure. Because C has finitely many configurations, this process eventually ends: applying the constructed schedule u to any configuration of C leads outside of C.

Therefore, for all n sufficiently large, $C[n]$ is included in the set of finite schedules of length n which do not contain u as a factor. In [14], the authors describe the asymptotic behaviour of the number of finite words of length n not containing a given word. Their results imply that there exists a constant $k > 0$, and a real value $1.7 < \theta < d$ (depending on u only) such that for all n $|C[n]| \leq k \cdot \theta^n + O((1.7)^n)$. In particular, $t(C) = \liminf_{n\to\infty} \frac{\log|C[n]|}{n\cdot\log d} \leq \frac{\log\theta}{\log d} < 1$. Since there are finitely many non-final components, we have $T_0 < 1$.

Final components reachability. Consider a T-random schedule S on G, and an input assignment α. Assume that the corresponding execution never reaches a final component of Γ. Then a suffix of this execution remains in some non-final component C forever. In particular, the underlying schedule S' of this suffix is such that, for all n, the prefix $S' \upharpoonright n$ belongs to $C[n]$. By Lemma 3, $H_G(S' \upharpoonright n) \leq \log|C[n]| + 4\log n + O(1)$. But S' is also T-random, which implies that

$$T \cdot n\log d \leq \log|C[n]| + 4\log n + O(1)$$
$$T \leq \frac{\log|C[n]|}{n\log d} + O\left(\frac{\log n}{n}\right)$$

Taking the inferior limit, we get $T \leq T_0$. Therefore, for every $T_0 < T \leq 1$, every execution whose underlying schedule is T-random reaches a final component.

Output is constant in any final component. Let α be an input assignment, and F any final component reachable from the initial configuration γ_α corresponding to α. We claim that the output assignments yielded by the configurations in F are all equal to some $\beta \in P(\alpha)$. Let S_0 be any finite schedule leading to F when applied to γ_α, and S be any 1-random extension of S_0 (it suffices to append a 1-random schedule, which exists by Lemma 4). Let E be the execution with schedule S starting with γ_α. The execution E eventually reaches the final component F and remains inside forever. Since $\mathcal{A}$ solves P under $\mathbb{A}(1, G)$, the output assignments associated with the configurations in F which are visited infinitely often during E are all equal to some output assignment $\beta \in P(\alpha)$. Since F is strongly connected, if the execution E did not visit all the configurations in F, then, by an argument similar to the second paragraph above, the schedule S could not be 1-random. Therefore, all the configurations in F are visited infinitely often during E, and they all yield the same output assignments $\beta \in P(\alpha)$.

Conclusion. Pick any $T_0 < T < 1$. Consider any input assignment α, and any T-random schedule S. The corresponding execution E reaches a final component F and remains in there forever. Moreover, the output assignments associated with the configurations in F are all equal to some $\beta \in P(\alpha)$. This implies that the execution E yields an output trace which is eventually constant and equal to $\beta \in P(\alpha)$. In other words, the algorithm $\mathcal{A}$ solves P under adversary $\mathbb{A}(T, G)$ with $T < 1$. In particular, the optimal entropy of P on G is less than 1. ◀

▶ Remark. Note that, the value $T_0 = T_0(\mathcal{A})$ defined in the proof is the optimal entropy above which the algorithm $\mathcal{A}$ solves the problem P on G. This is different, a priori, from the value $T(P, G)$ which is the optimal entropy above which, *some* algorithm solves the problem P on G.

5 Entropy Bounds for Leader Election

The previous section addressed the issue of randomness in a general setting. Now that we know that full randomness is not needed, it is natural to ask for the optimal entropy rate of a problem. We tackle this issue in this section, by considering the *leader election* problem.

Its specifications are the following. There is no input, and all the processes start in the same initial state and with the same initial knowledge. In particular, they do not have identifiers. The output of each process is 0 or 1. The goal is to eventually have a unique process permanently outputting 1 while the others permanently output 0. Note that, the processes are not required to make an irrevocable decision: we only ask for their outputs to eventually stabilize.

We are interested in computing upper and lower bounds on the optimal entropy rate for solving leader election. Actually, the approach taken in the proof of Proposition 7 already leads to an upper bound.

▶ **Proposition 8** (LE – upper bound)**.** *For any strongly connected graph G with $d = |\mathcal{E}(G)|$ edges, $T(LE, G) \leq \frac{\log \theta(G)}{\log d} < 1$ where $\theta(G)$ is the absolute value of the largest zero of the polynomial $1 + (z - d)(z^{2K(G)-1} + z^{K(G)-1})$, and $K(G)$ is the length of the shortest loop visiting each vertex at least once in G.*

We postpone the details to the appendix (Section B). The proof relies on an analysis of an algorithm from [8]. Roughly speaking, we exhibit a specific schedule S_0 of length $2K(G)$ which drives the system out of any non-final component of the transition graph. Then, we show that, whenever $T > \log \theta(G)/\log d$, any T-random schedule contains infinitely many occurrences of S_0.

As for a lower bound, the method presented in the proof Proposition 7 cannot be applied. In the sequel, we adopt another approach for deriving a lower bound.

5.1 Lower Bound

Our approach here exploits the fact that leader election requires symmetry breaking. If the communication graph G has local symmetries, then one can design schedules that maintain the symmetry of the system, as shown below. We prove that these schedules have an entropy rate, at most, $T_{sym}(G)$; thereby exhibiting a lower bound on $T(LE, G)$.

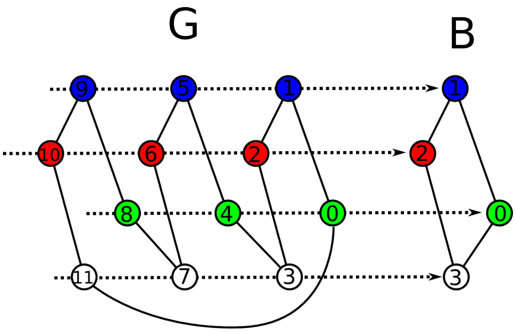


Figure 1 Graph covering: the ring G of 12 vertices is projected onto the ring B of 4 vertices, the dashed lines indicate the fibers, the degree is $12/4 = 3$.

Formally, the local symmetries of a graph G are measured by coverings. A *covering* is a surjective graph morphism $\phi : G \to B$ such that, for every vertex $b \in \mathcal{V}(B)$, for every neighbor c of b, for every vertex $x \in \phi^{-1}(b)$ (the *fiber over* b), there exists a unique neighbor y of x in the fiber over c. This concept measures the local symmetries of G in the sense that all vertices in the same fiber have isomorphic neighborhoods. It can be shown that every fiber $\phi^{-1}(b)$ has the same cardinality Δ_ϕ, and that $|\mathcal{E}(G)| = \Delta_\phi \cdot |\mathcal{E}(B)|$. The number Δ_ϕ is the *degree of the covering*. The covering is *proper* if $\Delta_\phi \geq 2$, i.e., ϕ is not an isomorphism.

▶ **Proposition 9** (LE – Lower bound). *For any graph G with $d = |\mathcal{E}(G)|$ edges*

$$T_{sym}(G) \underset{def}{=} \max_{\phi} \frac{\log(d/\Delta_\phi) + \log \Delta_\phi!}{\Delta_\phi \log d} \leq T(LE, G)$$

where ϕ runs over all the proper coverings from G. If there are no proper coverings, $T_{sym}(G)$ is set to zero.

To illustrate the proof's basic idea, fix any algorithm $\mathcal{A}$. Let's say that a configuration on G is symmetric if, any two processes in the same fiber (the same color in Figure 1) have the same state. Being uniform, the initial configuration is obviously symmetric. We describe a possible strategy to obtain symmetric configurations infinitely often. The adversary first selects an edge b of B, picks any enumeration of the fiber $\phi^{-1}(b)$ (the edges in G projecting to b), triggers the events according to the enumeration order, and repeats this operation. Since the algorithm $\mathcal{A}$ is deterministic, applying such a sequence of events to a symmetric configuration yields infinitely many symmetric configurations. The successive choices of the adversary are equivalently described by a sequence $Z = (b_1, \sigma_1)(b_2, \sigma_2)\dots$ where b_i is an edge of B, and σ_i is an ordering of the fiber $\phi^{-1}(b_i)$, i.e., an element of the permutation group $\mathfrak{S}_\Delta$ on $\{1, \dots, \Delta\}$. To maximize randomness, we assume that Z is 1-random on the alphabet $\mathcal{E}(B) \times \mathfrak{S}_\Delta$. This allows to compute the entropy rate $T_{sym}(G)$ of the schedule produced by the adversary given Z. Since this schedule prevents the election of a leader, $T_{sym}(G)$ is a lower bound of the optimal entropy rate $T(LE, G)$. Now, we present the full proof.

LE – **Lower bound.** Pick any proper covering $\phi : G \to B$ with degree Δ. We have $d = \Delta \cdot r$ where $r = |\mathcal{B}|$. For each edge b in B, we fix a reference enumeration of the fiber over b, $\phi^{-1}(b) = \{e_1(b), \dots, e_\Delta(b)\}$. We define $X = \mathcal{E}(B) \times \mathfrak{S}_\Delta$ where $\mathfrak{S}_\Delta$ is the group of permutations on $\{1, \dots, \Delta\}$. For each element $(b, \sigma) \in X$, we define the sequence $\psi(b, \sigma) = e_{\sigma(1)}(b) \dots e_{\sigma(\Delta)}(b)$ of edges in G; $\psi(b, \sigma)$ is simply an enumeration of the fiber $\phi^{-1}(b)$ ordered according to σ. Note that the map ψ is injective.

Let $\mathcal{A}$ be any (deterministic) algorithm. Assume that γ is a symmetric configuration in the sense that for every vertex $z \in B$, for any two vertices $x, y \in \phi^{-1}(z)$, $\gamma(x) = \gamma(y)$. Then for any element $\mu \in X$, applying the schedule $\psi(\mu)$ to γ yields a configuration γ' that is also symmetric.

Let $Z = \mu_1\mu_2\ldots$ be a 1-random sequence on X (which exists by Lemma 4). By definition, for all m, $H_X(Z \restriction m) \geq m \cdot \log(r \cdot \Delta!) - O(1)$. We define the schedule $S = \psi(Z) = \psi(\mu_1)\psi(\mu_2)\ldots$. Thanks to the previous remark, for any deterministic algorithm $\mathcal{A}$, since the initial configuration is symmetric (all the processes starts in the same state), the schedule S yields an execution in which, infinitely often, a symmetric configuration is reached. Hence, this prevents any algorithm to solve the leader election problem with the schedule S. Now, it remains to determine a lower bound on the entropy rate of S. For all n

$$S \restriction n = \psi(\mu_1)\ldots\psi(\mu_m)R$$

where $m = \lfloor n/\Delta \rfloor$. Each sequence $\psi(\mu_i)$ has length Δ, and R is strict prefix of $\psi(\mu_{m+1})$. Knowing $S \restriction n$ allows to compute the sequence $Z \restriction \lfloor n/\Delta \rfloor$. In other words, there exists a prefix-free recursive function $q : \mathcal{E}(G)^* \to \mathcal{E}(B)^*$ such that, for all n, $q(S \restriction n) = Z \restriction \lfloor n/\Delta \rfloor$. By Lemma 3, and the fact that Z is 1-random on X, we have, for all n

$$\begin{aligned} H_G(S \restriction n) &\geq H_X(Z \restriction \lfloor n/\Delta \rfloor) - O(1) \\ &\geq \frac{n}{\Delta} \cdot \log(r \cdot \Delta!) - O(1) \\ &\geq \underbrace{\frac{\log r + \log \Delta!}{\Delta \cdot \log d}}_{T(\phi)} \cdot n \cdot \log d - O(1) \end{aligned}$$

Therefore, S is $T(\phi)$-random; whence $T(\phi) \leq T(LE, G)$. ◀

▶ Remark. There is a subtlety about this lower bound related to the specification of the leader election problem we have chosen. Indeed, as stated above, the processes are not required to make an irrevocable decision, but only to have a stabilized output eventually.

When, instead, the processes have to make an irrevocable decision, the relevant notion of local symmetries (in the context of population protocols) is that of *pseudo-coverings*[3] [11]. Roughly speaking, a graph G is a pseudo-covering of another graph H, if there exists a subgraph G' of G with the same set of vertices such that G' is a covering of H. In the case where H is not isomorphic to G' (i.e., G is not pseudo-covering minimal), one can lift any terminating execution on H to a terminating execution on G such that, at the end, any process in G always has the same state as another process in G. In particular, leader election with irrevocable decision cannot be solved on a graph G which is not pseudo-covering minimal.

However, when irrevocable decisions are not required, the lifting argument does not hold. Indeed, one has to lift an infinite execution on H to an infinite execution on G in which *every* edge occurs infinitely often. This cannot be achieved if the intermediate graph G' is a strict subgraph of G. In other words, not being pseudo-covering minimal may not be an obstruction to the possibility of leader election without irrevocable decisions.

[3] We thank the reviewers for having highlighted this point.

6 Conclusion

We have shown that, once a problem can be solved by some finite-state algorithm for perfectly random (1-random) schedules, the optimal entropy rate of the problem is strictly less than 1. Doing so, we have exhibited a general method for computing upper bounds on a problem's optimal entropy rate. Next, we focused on the leader election problem. By refining the method above, we have computed an upper bound on the optimal entropy rate of leader election. Then, we computed a lower bound T_{sym} which encodes the maximum entropy rate of schedules maintaining symmetry during the execution. Notice that this lower bound also holds for any other problem requiring symmetry breaking like, e.g., enumeration or spanning tree construction.

This work opens many interesting questions. It seems that the bound T_{sym} could be reached by some algorithm with unbounded memory. The intuition goes as follows. The processes could record the whole history of their interactions, and try to deduce the past schedule. If this schedule breaks symmetry at some point, then it means that the processes have pairwise different "views" of the past. This distinction could be used to distinguish a leader among them. Nevertheless, the required memory may be unbounded, as the scheduler may maintain the symmetry for an arbitrarily long time. This leaves open the question of determining the optimal entropy rate of leader election achievable with finite-state algorithms.

In this work, we have focused on randomness in the scheduling. But one could also study algorithms involving local random coins. Similar questions can be raised, e.g., to what extent randomized algorithms are sensitive to imperfect local coins?

From a more general point of view, we believe that the relation between randomness and hardness of problems is not yet fully understood in the context of distributed computing.

References

1 Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994. doi:10.1006/inco.1994.1075.

2 Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 – June 03, 2014*, pages 714–723, 2014. doi:10.1145/2591796.2591836.

3 Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*, pages 82–93, 1980. doi:10.1145/800141.804655.

4 Dana Angluin, James Aspnes, Melody Chan, Hong Jiang, Michael J. Fischer, and René Peralta. Stably computable properties of network graphs. In *Distributed Computing in Sensor Systems*, pages 63–74. LNCS 3560, 2005.

5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.

6 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC'06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 292–299, New York, NY, USA, 2006. ACM Press. doi:10.1145/1146381.1146425.

7 James Aspnes. Fast deterministic consensus in a noisy environment. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'00, pages 299–308, New York, NY, USA, 2000. ACM. doi:10.1145/343477.343631.

8 Joffroy Beauquier, Peva Blanchard, and Janna Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *Principles of Distributed*

Systems – 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings, pages 38–52, 2013. doi:10.1007/978-3-319-03850-6_4.

9 Cristian S. Calude and Marius Zimand. Algorithmically independent sequences. In *Developments in Language Theory, 12th International Conference, DLT 2008, Kyoto, Japan, September 16-19, 2008. Proceedings*, pages 183–195, 2008. doi:10.1007/978-3-540-85780-8_14.

10 Gregory J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22(3):329–340, July 1975. doi:10.1145/321892.321894.

11 Jérémie Chalopin. Local computations on closed unlabelled edges: The election problem and the naming problem. In *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, pages 82–91, 2005.

12 Jérémie Chalopin, Yves Métivier, and Thomas Morsellino. Enumeration and leader election in partially anonymous and multi-hop broadcast networks. *Fundamenta Informaticae*, 120(1):1–27, 2012. doi:10.3233/FI-2012-747.

13 Yuval Emek, Christoph Pfister, Jochen Seidel, and Roger Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *ACM Symposium on Principles of Distributed Computing, PODC'14, Paris, France, July 15-18, 2014*, pages 96–105, 2014. doi:10.1145/2611462.2611478.

14 Leonidas J. Guibas and Andrew M. Odlyzko. String overlaps, pattern matching, and nontransitive games. *Journal of Combinatorial Theory, Series A*, 30(2):183–208, 1981. doi:10.1016/0097-3165(81)90005-4.

15 Alon Itai and Michael Rodeh. Symmetry breaking in distributive networks. In *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*, pages 150–158, 1981. doi:10.1109/SFCS.1981.41.

16 Andreï Nikolaïevitch Kolmogorov. Three approaches to the quantitative definition of information. *Problemy Peredachi Informatsii*, 1(1):3–31, 1965.

17 Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, Third Edition.* Texts in Computer Science. Springer, 2008. doi:10.1007/978-0-387-49820-1.

18 Per Martin-Löf. The definition of random sequences. *Information and Control*, 9(6):602–619, 1966.

19 Antoni Mazurkiewicz. Distributed enumeration. *Information Processing Letters*, 61(5):233–239, March 1997. doi:10.1016/S0020-0190(97)00022-7.

20 Ray J. Solomonoff. A formal theory of inductive inference part i. *Information and Control*, 7(1):1–22, March 1964.

21 Ray J. Solomonoff. A formal theory of inductive inference part ii. *Information and Control*, 7(2):224–254, June 1964.

22 Ludwig Staiger. The kolmogorov complexity of infinite words. *Theoretical Computer Science*, 383(2-3):187–199, 2007. doi:10.1016/j.tcs.2007.04.013.

23 Kohtaro Tadaki. A generalization of Chaitin's halting probability Ω and halting self-similar sets. *Hokkaido Mathematical Journal*, 31(1):219–253, 02 2002. doi:10.14492/hokmj/1350911778.

24 Kohtaro Tadaki. Partial randomness and dimension of recursively enumerable reals. In *Mathematical Foundations of Computer Science*, pages 687–699, 2009. doi:10.1007/978-3-642-03816-7_58.

25 Alexander K. Zvonkin and Leonid A. Levin. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys*, page 11, 1970.

A Interpretation of Lemma 3

We briefly give an intuitive interpretation of the facts mentioned in Lemma 3.

▶ **Lemma 3** (restated).

(a) *Let $q : X^* \to Y^*$ be a partial recursive function. Then, for all $S \in X^*$, $H_Y(q(S)) \leq H_X(S) + O(1)$.*

(b) *For all $S \in X^*$, $H_X(S) \leq |S| \cdot \log |X| + 2 \cdot \log |S| + O(1)$.*

(c) *Let $A \subseteq X^* \times \mathbb{N}$ be a recursively enumerable set such that the subset $A_n = \{w : (w, n) \in A\}$ is finite for every $n \in \mathbb{N}$. Then, for all n, for every $S \in A_n$ with $|S| = n$, $H_X(S) \leq \log |A_n| + 4 \cdot \log n + O(1)$.*

(d) *An infinite sequence $S \in X^\omega$ is T-random if and only if any of its suffixes is T-random.*

The point *a.* states that the information content of the sequence $q(w)$ is no more than the information content of w (up to an additive constant). This simply comes from the fact that a computer program cannot produce more information than the information already present in its input.

The point *b.* comes from the fact that one can define a computer program p which simply copies its input to its output. Therefore, a program for generating the sequence S is given by the concatenation $\langle p, S\rangle$ of the program p, and the input S itself. This concatenation has length $|S|$ plus the length of p (which is independent of S), and an additional logarithmic term required to distinguish the two sequences p, S in the concatenation. The inequality follows by the definition of the complexity as the length of the shortest program producing the sequence S.

The interpretation of *c.* is slightly more involved. The set A being recursively enumerable means that there is a computer program p which can successively enumerate all the elements of A. Thanks to p, one can design a new program q which takes as input a number n (requiring $\log n$ bits), and an index $1 \leq i \leq |A_n|$ (requiring $\log |A_n|$ bits), and returns the i-th element in A_n having length equal to n. Therefore an element $S \in A_n$ with $|S| = n$ can be generated by the program $\langle q, i, n\rangle$ where i is the corresponding index of S. This explains the logarithmic terms on the right hand side.

Finally, the point *d.* simply comes from the fact that the notion of T-randomness depends only on the asymptotic behaviour of the infinite schedule.

B Leader election – upper bound

The proof is achieved by analyzing a specific algorithm, hereafter called $\mathcal{B}^3$, from a previous work [8]. The authors have shown that the algorithm $\mathcal{B}^3$ solves leader election over arbitrary strongly connected graphs using another fairness assumption, namely the global fairness [5]. We show that, actually, the algorithm $\mathcal{B}^3$ solves leader election on G under adversary $\mathbb{A}(T, G)$ for every $T > \log \theta(G)/\log d$. In the sequel, we use some combinatorial results from [14].

▶ **Definition 10** (Word Correlation [14]). Let u, v be two sequences in X^*. The correlation $\langle u, v\rangle$ is the polynomial $a_1 z^{|u|-1} + a_2 z^{|u|-2} \cdots + a_{|u|}$ where $a_i \in \{0, 1\}$ is obtained as follows. Place v under u so that its leftmost symbol is under the i-th symbol of u. Then if all the pairs of symbols in the overlapping segment are identical, $a_i = 1$, else $a_i = 0$.

▶ **Lemma 11** (Number of words omitting some word [14]). *Consider some sequence $u \in X^*$ of length k. Let $f(z) = \langle u, u\rangle$ be the autocorrelation polynomial of u. Then, the absolute value*

θ_u of the largest zero of the polynomial $1 + (z - d)f(z)$ satisfies $1.7 < \theta_u < d$. Moreover, the set $A_n(u)$ of sequences of length n not containing u as a factor satisfies

$$|A_n(u)| = \frac{\theta_u^n}{1 - (d - \theta_u)^2 f'(\theta_u)} + O((1.7)^n)$$

Now, we prove that the algorithm $\mathcal{B}^3$ presented in [8] solves leader election on G with adversary $\mathbb{A}(T, G)$ for all $T \in (\log \theta(G)/\log d, 1]$.

The pseudo-code is presented in Algorithm 1. Each process x can be leader or non-leader (variable $leader_x$) and can hold a white or black token (variable $token_x$). Initially, every process is a leader and holds a black token. The tokens move through the network by swapping between two processes during an interaction. When two black tokens meet, one of them turns white. When a white token interacts with a leader x, x becomes a non-leader and the token is destroyed. Given a configuration γ, let $b(\gamma)$ be the number of black tokens, $w(\gamma)$ the number of white tokens and $l(\gamma)$ the number of leaders in γ.

Algorithm 1: Algorithm $\mathcal{B}^3$

```
variables for every process x:
    $leader_x$ : 0 (non-leader) or 1 (leader);
    $token_x$ : ⊥ (no token), white or black;
initialization: $\forall x, (leader_x, token_x) = (1, black)$;              /* uniform */
algorithm (initiator x, responder y):
    if $token_x = token_y = black$ then
        $token_y \leftarrow white$;
    if $token_x = white \wedge leader_y = 1$ then
        $leader_y \leftarrow 0$ ;                              /* y becomes a non-leader */
        $token_x \leftarrow \perp$ ;                              /* the token is destroyed */
    $token_x \leftrightarrow token_y$;                                     /* swap the tokens */
```

▶ **Lemma 12.** *For any configuration γ of Algorithm 1 reachable from the initial configuration, $b(\gamma) + w(\gamma) = l(\gamma)$ and $b(\gamma) \geq 1$.*

Proof. The initial configuration satisfies this relation. During an interaction, if no leader is turned into a non-leader, then the total number of tokens remains constant. When a leader is turned into a non-leader (by a white token), the corresponding token is also destroyed. Moreover, destroying a black token requires that another black token collide with it, so there is always at least one black token. In any case, the first formula still holds. ◀

Consider the shortest loop $\pi = (a_1, a_2)(a_2, a_3)\dots(a_k, a_1)$ in G which visits every node at least once. We have $k = K(G)$. We define a finite schedule $S_0 = \pi\pi$, i.e., the path π repeated twice.

▶ **Lemma 13.** *Let γ be any configuration reachable from the initial configuration such that $l(\gamma) \geq 2$. Then, applying the finite schedule S_0 to γ yields a configuration γ' such that $(1, 1) \leq (l(\gamma'), b(\gamma')) < (l(\gamma), b(\gamma))$ in the lexicographical order (l first).*

Proof. We denote by l, b, w (resp. l', b', w') the value of $l(\gamma), b(\gamma), w(\gamma)$ (resp. $l'(\gamma), b'(\gamma), w'(\gamma)$). We first examine the case $b = 1$. By Lemma 12, we have $w = l - 1 \geq 1$ and, since no black token is ever created, $b' = 1$. Let a_i be some process in the path π which holds a white token in the configuration γ. When applying the schedule S_0 to the configuration γ, the white token moves (by swapping) from the process a_i through all the processes of G at least once. Thus, the white token necessarily meets with some leader in the graph, and this leader then disappears (along with the white token). Therefore, $l' < l$. Assume now that $b \geq 2$. When applying the schedule S_0 to γ, either the previous scenario occurs (and then $l' < l$), or two black tokens meet, and one of them turns white (thus $b' < b$). In either case, we have $(l', b') < (l, b)$. ◀

▶ **Proposition 14.** *For any strongly connected graph G*

$$T(LE, G) \leq \frac{\log \theta(G)}{\log d} < 1$$

where $\theta(G)$ is the absolute value of the largest zero of the polynomial $1 + (z - d)(z^{2K(G)-1} + z^{K(G)-1})$ with $K(G)$ being the length of the shortest loop visiting every node at least once in G.

Proof. Consider any $T > \log \theta(G)/\log d$, and any T-random schedule S. We prove that S_0 occurs infinitely often in S. Assume that S_0 occurs only finitely many times. Without loss of generality, we can assume that S_0 does not occur at all in S. Then, for every n, the prefix $S \restriction n$ belongs to the set $A_n(S_0)$ of finite schedule of length n which do not contain S_0 as a factor. By Lemma 3, we have, for all n,

$$H_G(S \restriction n) \leq \log |A_n(S_0)| + 4 \log n + O(1) \tag{1}$$

To estimate the value of $|A_n(S_0)|$, we compute the autocorrelation polynomial of $S_0 = \pi\pi$. Since π is a loop of shortest length $k = K(G)$ visiting every node at least once, the autocorrelation polynomial of π is $z^{K(G)-1}$. Indeed, let $\pi = (a_1, a_2) \dots (a_k, a_{k+1})$ with $a_{k+1} = a_1$. Assume that for some $1 \leq i \leq k$,

$$(a_i, a_{i+1}) \dots (a_k, a_1) = (a_1, a_2) \dots (a_{k-i+1}, a_{k-i+2})$$

Then, defining $j = \max(i, k - i + 1)$, the sequence $(a_1, a_2) \dots (a_j, a_{j+1})$ is a loop visiting every vertex at least once. This imposes $j = k$, and thus $i = 1$. By definition, this implies that the autocorrelation polynomial of π is $z^{K(G)-1}$. Since $S_0 = \pi\pi$, the autocorrelation polynomial of S_0 is $z^{2K(G)-1} + z^{K(G)-1}$.

Note that $\theta = \theta(G)$ is precisely the absolute value of the largest zero of the autocorrelation polynomial of S_0. By Lemma 11, there exists a constant $k > 0$ such that, for all n, $|A_n(S_0)| = k\theta^n + O((1.7)^n)$. Plugging this into Equation 1, and using the fact that S is T-random, we have, for all n,

$$T \cdot n \log d \leq n \log \theta(G) + 4 \log n + O(1)$$

Therefore, $T \leq \log \theta(G)/\log d$; whence a contradiction. This proves that S_0 occurs infinitely often in S.

Then, by Lemma 13, the corresponding execution of $\mathcal{B}^3$ satisfies the specification of the leader election problem. In other words, $\mathcal{B}^3$ solves leader election with adversary $\mathbb{A}(T, G)$. In particular, this shows $T(LE, G) \leq T$ for every $T > \log \theta(G)/\log d$; whence $T(LE, G) \leq \log \theta(G)/\log d$. ◀

▶ Remark. Note that this algorithm was designed in [8] to solve leader election without any knowledge about the underlying communication graph. Adding such a knowledge may yield a better upper bound on $T(LE, G)$.

Byzantine Agreement with Median Validity

David Stolz[1] and Roger Wattenhofer[2]

1 ETH Zürich, Zürich, Switzerland
stolzda@ethz.ch
2 ETH Zürich, Zürich, Switzerland
wattenhofer@ethz.ch

Abstract

We introduce a stronger validity property for the byzantine agreement problem with orderable initial values: The *median validity property*. In particular, the decision value is required to be close to the median of the initial values of the non-byzantine nodes. The proximity to the median scales with the desired level of fault-tolerance: If no fault-tolerance is required, algorithms have to decide for the true median. If the number of failures is maximal, algorithms must still decide on a value within the range of the input values of the non-byzantine nodes. We present a deterministic algorithm satisfying this property for $n \geq 3t + 1$ within $t + 1$ phases, where t is the maximum number of byzantine nodes and n is the total number of nodes.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Reliability, fault-tolerance, median, consensus

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.22

1 Introduction

A distributed system consists of a number of nodes, which collaboratively achieve fault-tolerance: Even if some of the nodes fail, the system as a whole can continue to operate reliably. As such, distributed systems are omnipresent in areas that have strong availability and reliability requirements. The challenging problem is to keep the system in a consistent and reasonable state. The following example illustrates this problem.

An airplane control system consists of multiple machines. In order to reach a decision on what action the plane performs next, the system relies on data from various sensors. One type of sensor is the *altitude meter*. Assume that the system needs to decide whether or not it is safe to descend, and queries the altitude meters. As those sensors are susceptible to freezing – and known to malfunction when frozen – each plane is equipped with multiple altitude meters. The airplane control system needs to agree on a value, and in order to avoid a single point of failure, this agreement will be achieved in a distributed manner.

One may argue that this agreement operation is at the core of many distributed systems, and generally fundamental for distributed computing research. Depending on the failure model, it is sometimes referred to as *consensus* (to cope with crash failures, e.g. [10]) or as *byzantine agreement* (to cope with arbitrary failures, e.g. [24]). In our airplane scenario – as well as in many other scenarios – one must be able to tolerate arbitrary failures, as a frozen sensor might not simply crash but still continue to provide data, even though that data is totally bogus. Nodes not complying with the protocol or reporting wrong data are called byzantine, nodes which behave correctly we simply call good.

The byzantine agreement (BA) problem consists of three properties: Termination, agreement, and validity. We need termination since we require our algorithm to come to a decision within finite time, as not deciding how high a plane is flying for a long time is disastrous. We

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 22; pp. 22:1–22:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

need agreement as we want the nodes of our distributed system to continue their operation in the same state. To rule out trivial solutions ("The altitude is always 10,000 meters.") we need a third property: Validity. The purpose of the validity property is to make sure that the value agreed on is reasonable. There exist slightly different definitions for validity, but the most common one is the following (see, e.g. [12, 4]):

Validity: If all good nodes start with the same input value, they agree on this value.

Unfortunately, there is no limitation on what the decision value can be for configurations in which not all good nodes start with the same input value. Surprisingly, in the literature, dissenting configurations receive only little attention. Informally speaking, one is obliged to accept a solution as long as the nodes agree on *any value*. The rationale for this weak validity condition is as follows: Since initially there is disagreement among the good nodes, a byzantine node can propose a bad value but follow the protocol of the algorithm, and there is no way to guarantee that the decision value will always be the input value of a good node. While this argument is formally correct, it implicitly assumes that it is impossible to distinguish good from bad values.

We believe that there is room for improvement, in particular in cases where input values can be ordered. Assume that we use an existing algorithm for BA in the previously described airplane example, and that we have four altitude meters which report the altitudes 995, 1002, 1004, and 5000, respectively. Even though it is clear that the correct altitude is roughly 1000, existing algorithms for BA might choose 5000, which might lead to a catastrophe. Often one must solve BA on input values in $\mathbb{R}$, e.g., measurements. Measured values are usually prone to noise, hence even the good nodes will have slightly different input values. The classic validity property (which is only concerned with the case where all input values are equal) does not help. To address this problem, we introduce a stronger validity property: The *median validity property*. In particular, we require the decision value to be "reasonably close" to the median of the values of the good nodes. We will later specify this requirement precisely.

In Section 4 we present a new BA algorithm called the Jack algorithm. Our algorithm has an optimal resilience, i.e., it works as long as $n \geq 3t + 1$, where n is the total number of nodes, and t denotes the *tolerance*, i.e., the maximum number of byzantine nodes. Note that we distinguish between t and f, where $f \leq t$ is the actual number of byzantine nodes during an execution of the algorithm, analogous to [8]. This distinction is necessary, as the nodes only know the tunable parameter t, since f is only known to the adversary. We will prove that the algorithm works for every $f \leq t$, thus the algorithm does not rely on any particular f. The Jack algorithm is deterministic and terminates in $t + 1$ phases, where each phase consists of a constant number of synchronous rounds. The algorithm additionally guarantees that the decision value will be reasonably close to the median of the good nodes. The proximity of the decision to the median of the good nodes is defined in an elastic way: In the best case, if we do not need any fault-tolerance, the algorithm agrees on the median itself. In the worst case, if the number of byzantine nodes is maximal, we require the value only to be within the interval spanned by the good nodes; in our airplane example, the algorithm will choose a value x with $995 \leq x \leq 1004$.

In addition, we present a lower bound for our validity property. If all byzantine nodes collude and suggest arbitrarily small (respectively large) input values, no algorithm can guarantee to find the correct median of the good nodes. In Section 5, we will provide exact bounds on this error.

2 Related Work

The BA problem has a long history, starting in 1980 when it was first introduced by Lamport et al. [24]. Since then, a vast amount of research has been conducted and many models and questions have been studied. The earlier works focused mainly on general computability questions, e.g.,[10, 2]. The first algorithms required message sizes exponential in n, or had a runtime that was far from optimal, but these shortcomings have been addressed and solved later.

With the proliferation of data centers, companies started implementing a large number of BA protocols, and the interest from the practical distributed systems community exploded. Well-known protocols include e.g. PBFT [6], Farsite [1], Google's Chubby [5], PeerReview [14], Zyzzyva [17], AZyzzyva [13], Apache ZooKeeper [15], SMART [20], ZZ [27], or RAFT [22]. These protocols try to improve various practical aspects of BA, in particular runtime, but do not address validity. We believe that some of these protocols could benefit from our orthogonal goal of strengthening the validity property.

Our contribution is a relative of the Queen [3] and King [4] algorithms by Berman, Garay (and Perry). These algorithms operate in the synchronous model and terminate in $t+1$ phases, which is asymptotically optimum for algorithms that terminate *simultaneously* [8]. While the Queen algorithm does not have optimal resilience, the King algorithm tolerates the maximum number of byzantine nodes. Our Jack algorithm achieves optimum runtime and resilience as well, and additionally satisfies a stronger validity condition as discussed in the introduction.

There are also algorithms that are closely related to the BA problem, but do not satisfy all three BA properties. In *distributed average consensus*, nodes try to achieve consensus on the average of all input (sensor) values. There are some major differences to BA: First, algorithms do not tolerate byzantine failures, but, e.g., only link failures [29, 23], systematic failures on the input (noise) [28], or both [16]. Second, the goal is not to agree on a specific value after a finite amount of time, but rather to *converge* towards a value. Hence, these algorithms do not satisfy the termination property, and cannot be used in many scenarios, e.g. in our airplane example. Please refer to the book of Ren and Beard [25] for an extensive survey.

The *clock synchronization problem* [18] is another agreement problem, which usually assumes that the clock values are initially close to each other, however, not exactly equal. While this problem was only loosely related to the BA with the classical validity property, the relation to BA with median validity is evident, since an agreement on the median is a reasonable goal for clock synchronization.

In a very recent work [26] Su and Vaidya study the problem of byzantine distributed optimization of real valued convex cost functions. Instead of starting with an initial value, every node starts with a cost function $h(x)$. The goal is to reach consensus on a value x, such that the total cost (the sum of the costs of the good nodes) is minimized. Since the optimization problem is similar to the problem we study, it is not surprising that they establish a proof that the optimization problem cannot be solved *exactly* in the presence of byzantine nodes, which conveys a similar message as our lower bound (see Section 5).

The work which is most closely related to our contribution is the one which aims at strengthening the validity property. We are aware of two different approaches. The first one is the *approximate agreement problem* by Dolev et al. [7]. Dolev et al. strengthen the validity property at the expense of the agreement property. The agreement property by Dolev et al. does not require the nodes to agree consistently on one value, but allows the nodes to choose different values which are in ε-distance of each other. In their algorithm, the range of acceptable values is iteratively reduced, until the range satisfies the ε-property. They can guarantee that the algorithm terminates as early as possible. On the other hand, the agreement property cannot be guaranteed anymore. We believe that many applications depend on all three BA properties, as future actions depend on past decision values.

The second one is the *strong consensus problem* [21, 11]. The strong consensus problem allows input values from a finite Domain $\mathcal{D}$, and strengthens the validity property by requiring the nodes to agree on a value that was the initial value of at least one good node. It might seem that such a validity property is also the right choice for real valued inputs; however, Neiger [21] showed that n must be at least $|\mathcal{D}| \cdot t$, which is clearly not applicable to scenarios which require initial values in $\mathbb{R}$.

Our validity property is hence both in the spirit of Dolev et al.[7] and Neiger [21], and we think that we found a reasonable trade-off between agreement, validity of the decision and the required number of nodes.

3 Model

Let A denote an array of length l. Throughout this paper we will use zero-based arrays, meaning that the first element of an array is $A[0]$ and the last element is $A[l-1]$. When we refer to the *median* of an array A, we want the median to be a value which actually *occurs* in that array. As this raises a problem with arrays of even size, we define the median to be the element at index $A[\lceil \frac{l}{2} \rceil - 1]$, assuming A is sorted.

The network consists of n nodes which are fully connected, i.e., every node can communicate with every other node. Nodes communicate by exchanging messages, and we assume that time elapses in *synchronous rounds*. We allow every node to send exactly one message to every other node in each round, and we assume that these messages will be received in the following round. The network itself is assumed to be reliable, meaning that messages are not altered or delayed between sender and receiver. The different links between nodes are isolated from each other, i.e., only the sender and receiver of a message know about the message. Hence, if a node *broadcasts* a message, it sends a single message to every node, including itself.

The byzantine agreement problem tends to be specified in different ways in different papers: Sometimes the input values are only binary, sometimes there is a predetermined “leader node”. In this paper, we define the BA problem as follows:

Every node has an input value $x \in \mathbb{R}$. There are $f \leq t$ byzantine nodes which can behave *arbitrarily*. This includes lying about their input value, not following the protocol, sending different messages to different nodes in the same round, crashing, and so on. We allow an *adaptive* adversary, which can determine during the execution of the protocol which nodes are byzantine, and how they behave. Note however, that the adversary is limited to selecting at most t nodes, meaning that a node that is byzantine cannot be declared as non-byzantine at later point in time. We call a node *good* if it is not selected by the adversary to be byzantine at any point throughout the execution; it follows directly that there are always at least $n-t$ good nodes. However, a good node does not a priori know *which* of the other nodes are good; a good node only knows that there must be at least $n-t$ good nodes in the system. At the end of the algorithm, every good node must have decided for a certain value, which we will refer to as the *decision*.

Any algorithm that solves byzantine agreement with median validity must satisfy the following three properties:

Agreement: For every selection of input values and every selection of byzantine nodes, all good nodes must decide on the same value.
Termination: Every good node must decide on a value in finite time.
Median Validity: The decision is always a *valid value*[1] (see definition below).

Let us now define which values are valid. For that purpose, we denote by G the *sorted* array of the input values of the good nodes. Note that the length of the array is $n - f \geq n - t$, and the median of the good nodes is $G[\lceil \frac{n-f}{2} \rceil - 1]$.

▶ **Definition 1** (valid value)**.** We call a value x *valid*, if it holds that

$$G[\left\lceil \frac{n-f}{2} \right\rceil - 1 - t] \leq x \leq G[\left\lceil \frac{n-f}{2} \right\rceil - 1 + t].$$

Note that in the best case, where $f = t = 0$, the median of the good nodes is the only valid value. In the worst case, where $n = 3t + 1$ and $f = t$ (which corresponds to the maximum number of byzantine nodes so that BA is solvable, see [24, 19]), every value which is between the smallest input value of a good node and the largest input value of a good node is valid.

Observe that even though the decision must be between the smallest and largest input value of a good node, the decision is *not* necessarily the input value of a good node! For example, we allow that the value proposed by a byzantine node can be chosen, as long as it satisfies the required proximity to the median. Even though this relaxation has only little negative impact on the quality of the decision, it is a crucial enabler of the validity property: As shown by [11], a validity property which also satisfies that the decision is the input value of a good node is impossible for real values, and even with input values from a finite domain, the failure tolerance decreases linearly with the size of the domain, which is an unfeasible requirement for many applications.

4 The Jack Algorithm

We developed an algorithm that uses some ideas from the *Phase King algorithm* developed by Berman, Garay and Perry [4]. One such idea is to have $t + 1$ nodes predetermined as *kings*. In $t + 1$ phases there is always one of those kings the depicted king for that phase, assuming a special role. We use the same idea in our algorithm, but refer to this special role as the *jack*.

Note that the Jack algorithm requires the tolerance as an input parameter `t`, which is tunable. The choice of `t` affects the runtime of the algorithm, its fault-tolerance, and how close to the median of the good nodes the final decision value will be. In particular, `t` determines the maximum number of byzantine nodes that the algorithm can tolerate, and it determines which values are considered valid, according to the definition of a valid value (see Definition 1).

All nodes start the execution of the algorithm at the same time, and execute the same steps at the same time; the only exception is the step marked with **Only Jack**, in which only the respective jack of that phase is active, and all other nodes remain silent. Note that since it is predetermined which node is the jack of which phase, every node can simply ignore byzantine nodes pretending to be the jack in a wrong phase.

[1] Note that the classic validity property is implied by our validity property as a special case.

Algorithm 1: The Jack algorithm

```
input  : inputValue, n, t
output : consensusValue

Setup Stage

Broadcast(inputValue)

Store and sort all received values in array values
interval = values.Subarray($\lceil \frac{n-t}{2} \rceil - 1$, $n - \lfloor \frac{n-t}{2} \rfloor - 1$)
Broadcast("bounds: interval.first, interval.last")

Store all received interval bounds
if inputValue is in at least n-t bounds then
    suggestion = inputValue
else
    suggestion = pick any value from interval that is in at least n-t bounds
current = suggestion

Search Stage

for i = 1 to t + 1 do
    Broadcast(current)

    if some value x appears $\geq$ n-t times then
        Broadcast("propose x")

    if some "propose x" received $>$ t times then
        current = x

Only Jack
    if some "propose x" received $>$ t times then
        jackSuggestion = current
    else
        jackSuggestion = suggestion
    Broadcast("suggest jackSuggestion")

    if current == jackSuggestion or
    (interval.first $\leq$ jackSuggestion $\leq$ interval.last) then
        Broadcast("support jackSuggestion")

    if received "propose x" $<$ n-t times then
        if some "support jackSuggestion" received $>$ t times then
            current = jackSuggestion

consensusValue = current
```

The Jack algorithm works as follows: At the start there is a *setup stage* consisting of 3 rounds, in which all nodes determine which values they will support later on. It is followed by the *search stage* consisting of t+1 phases, in which the nodes agree on the decision value. In each of these phases there is at first a part where all nodes perform the same actions, and a second part, in which a predetermined jack tries to suggest a value. As there are t+1 jacks, we are guaranteed to have at least one non-byzantine jack. Since we do not know in which of these phases we have a non-byzantine jack, we need the first part: This part prevents a byzantine jack to change an existing agreement. The second part (the part of the jack) allows a good jack to suggest a valid value, so that we are assured to reach agreement.

In the following sections, we prove the correctness of the algorithm.

4.1 Preliminaries

▶ **Lemma 2.** *The median of the good nodes $G[\lceil\frac{n-f}{2}\rceil - 1]$ is in the* interval *of every good node.*

Proof. The intervals are created during the setup stage in Step 2 by truncating the array of received input values on both sides, i.e., by removing a subarray with the smallest values, and by removing a subarray with the largest values. In order to show that the median of the good nodes will always remain in the subarray interval, we show that it cannot be in either of two parts of the array that are removed.

Let us first look at the part of the array which is removed since the values within this part are considered too small. The smallest index of the sorted array that is kept in the array interval is $\lceil\frac{n-t}{2}\rceil - 1$. Since we use arrays that start at index 0, the number of values that are smaller or equal to the value at a certain index of the array is equal to the index. Thus, we know that the construction of the subarray interval discards the smallest $\lceil\frac{n-t}{2}\rceil - 1$ values. As there are $\lceil\frac{n-f}{2}\rceil - 1$ input values of good nodes which are smaller than the median of the good nodes (by definition of the median), and since $\lceil\frac{n-f}{2}\rceil - 1 \geq \lceil\frac{n-t}{2}\rceil - 1$, it follows that the median of the good nodes will never be removed because it is considered too small.

Analogously we show that the median of the good nodes cannot lie in the part of the array that is removed due to the values being too large. The largest index of the received value that is kept is $n - \lfloor\frac{n-t}{2}\rfloor - 1$. Therefore, the number of received values that are discarded since they are considered too large is $\lfloor\frac{n-t}{2}\rfloor$. Again, due to definition of the median, this is at most the number of input values of good nodes which are larger or equal to the median of the good nodes. Thus, the median of the good nodes will also never be considered to be too large, which concludes the proof. ◀

▶ **Lemma 3.** *All values stored in the* interval *arrays of good nodes are valid.*

Proof. Note that valid values are around the median of the good nodes. Thus, for a value to be not valid, it must either be too small, or too large. We prove the lemma by proving the two cases individually.

The smallest value of the received input values that is taken into the interval is at the index $\lceil\frac{n-t}{2}\rceil - 1$ of the sorted array of all received values, thus the number of input values that are discarded since they are too small is $\lceil\frac{n-t}{2}\rceil - 1$. Note that $f \leq \lceil\frac{n-t}{2}\rceil - 1$, since:

$$\begin{aligned} 3t &< n && \mid -t, \cdot\frac{1}{2} \\ t &< \frac{n-t}{2} \\ t &\leq \left\lceil\frac{n-t}{2}\right\rceil - 1 && \mid f \leq t \\ f &\leq \left\lceil\frac{n-t}{2}\right\rceil - 1 \end{aligned}$$

Within those values that are discarded, there are at most f values from byzantine nodes. Therefore, the number of good values that are discarded is at least $\lceil\frac{n-t}{2}\rceil - 1 - f$, meaning that the largest input value of a good node that is discarded because it is too small is at least $G[\lceil\frac{n-t}{2}\rceil - 1 - f - 1]$.

We can distinguish two cases: The first case is that all the byzantine nodes have sent a value that is discarded since it was too small. In this case, the smallest value in the interval must be from a good node, and since $\lceil\frac{n-t}{2}\rceil - 1 - f$ good values have been discarded, the smallest value of the interval is $G[\lceil\frac{n-t}{2}\rceil - 1 - f]$, which is at least as large as the lower bound of being valid, since:

$$\begin{aligned} f &\leq t \\ \left\lceil\frac{n-f}{2}\right\rceil + f &\leq \left\lceil\frac{n-t}{2}\right\rceil + t && \mid -f - t - 1 \\ \left\lceil\frac{n-f}{2}\right\rceil - 1 - t &\leq \left\lceil\frac{n-t}{2}\right\rceil - 1 - f \end{aligned}$$

And therefore $G[\lceil\frac{n-f}{2}\rceil - 1 - t] \leq G[\lceil\frac{n-t}{2}\rceil - 1 - f]$.

The second case is where at least one byzantine node sent a value that is not discarded for being too small. In that case, the number of good values that are discarded for being too small is at least one more than in the previous case. In particular, the value $G[\lceil\frac{n-t}{2}\rceil - 1 - f]$ will be discarded. Since all nodes that are not discarded must be bigger than the discarded ones, the lower bound holds.

We show that no value in the interval can be too large to be valid analogously. The number of values that are not taken into the interval since they are too large is $\lfloor\frac{n-t}{2}\rfloor$. As there might be at most f values from byzantine nodes, there will be at least $\lfloor\frac{n-t}{2}\rfloor - f$ values from good nodes discarded because they are too large. Hence, the smallest discarded good value is at most $G[(n-f) - (\lfloor\frac{n-t}{2}\rfloor - f)] = G[\lceil\frac{n-t}{2}\rceil + t]$. We distinguish the two cases again: First, the case where all byzantine values are discarded, as they are too large. In that case, the largest value kept in the interval is $G[\lceil\frac{n-t}{2}\rceil + t - 1]$, and since $\lceil\frac{n-t}{2}\rceil \leq \lceil\frac{n-f}{2}\rceil$, it follows that $G[\lceil\frac{n-t}{2}\rceil + t - 1] \leq G[\lceil\frac{n-f}{2}\rceil + t - 1]$, i.e., the required bound is satisfied. Second, the case where there is at least one byzantine value that is not discarded. In that case, the value $G[\lceil\frac{n-t}{2}\rceil + t - 1]$ is discarded, implying that all values in the interval are smaller than this value, and thus the upper bound holds. ◀

4.2 Main Theorem

▶ **Theorem 4.** *The* Jack algorithm *achieves byzantine agreement with median validity in* $O(t+1)$ *rounds, if and only if* $n \geq 3t+1$.

Note that the Jack algorithm matches the upper bound on the number of byzantine nodes that it can tolerate [19], and that it has an asymptotically optimal runtime [8, 9].

Proof. To facilitate the readability of the proof, we split it up into separate lemmas. Note that throughout the proof, we often use the fact that $n - f \geq n - t$ implicitly.

▶ **Lemma 5.** *Each good node has at least one value in its* interval *which is within the bounds of at least $n - f$ received bounds.*

Proof. We proved that the median of the good nodes is in the interval of every good node (Lemma 2). Hence the value of the median of the good nodes is at least in all good bounds, i.e., every good node receives at least $n - f$ bounds containing the median of the good nodes. Since every good node has the median in its own interval, the lemma holds. ◀

This lemma guarantees that the setup stage can always be completed successfully, i.e., that every good node can always find an eligible suggestion.

▶ **Lemma 6.** *If all good nodes have the same value stored in the variable* current *at the beginning of a phase, they will not change the value of* current *during this phase.*

Proof. Let v denote the value which is stored in the variable current at all good nodes. In Step 4 all $n - f$ good nodes broadcast v, and thus all good nodes hear at least $n - f$ messages containing v. Therefore, all good nodes propose v in Step 5, and all good nodes hear at least $n - f$ proposals for v in Step 6. They update their value to v, i.e., they keep v. The only other step where the nodes update their variable current is Step 8, but since every good node received at least $n - f$ proposals for v, no good node updates to the jackSuggestion. ◀

▶ **Corollary 7.** *If all good nodes have the same value v stored in the variable* current *in the beginning of any phase, they will agree on v when the algorithm terminates.*

▶ **Lemma 8.** *If all good nodes start with the same input value v, they will agree on this value. (The classic validity property holds.)*

Proof. Since all good nodes start with the input value v, this value is also the median of the good nodes. Lemma 2 states that v must be in the interval of every good node. Thus, every good node sets its variable current to v in Step 3. As every good node starts the search stage with current set to v, it follows from Corollary 7 that they agree on v when the algorithm terminates. ◀

▶ **Lemma 9.** *A good node will only store values in its variable* current *which are within the bounds of an* interval *of at least one good node.*

Proof. Note that every good node picks a value from its own interval as the starting value for current, thus the lemma holds at the end of the setup stage. For the search stage, it is sufficient to show, that if the lemma holds at the beginning of a phase, it will also hold at the end of the phase (and thus at the beginning of the next phase). There are two steps in which a good node can update its variable current: Step 6 and Step 8.

If a good node updates the variable current in Step 6 to a new value x, we know that more than t nodes sent a proposal for x, i.e., at least one good node sent a *"propose x"*. Hence, the good node which sent the proposal must have received x in Step 5 least $n - t \geq 2t + 1 > f$ times, meaning that at least one good node must have sent x in Step 4. Since we assumed that the lemma holds at the beginning of the phase, this value must have been in the interval of a good node.

The second possibility to update current is in Step 8. If a good node updates the value in Step 8 to a jackSuggestion x, it must have received a message *"support x"* from at least one good node. Such a message is sent in Step 7, but a good node only supports a suggestion from the jack if the suggestion is either the current of the good node, or if the suggestion is within the bounds of its interval. In the latter case the lemma holds, as this is condition enforces exactly what the lemma states. Looking at the former case, we know that the lemma holds after Step 6, and thus it also holds in Step 7, as current was not changed in between. Therefore, x must be within the interval of at least one good node, concluding the lemma. ◀

Combining this lemma with Lemma 3 yields the following corollary:

▶ **Corollary 10.** *A good node always has a valid value stored in its variable* current*.*

Since the value for which a good node decides when the algorithm terminates is the value stored in the variable current, it follows that every good node will always decide for a valid value. Before we can establish the last lemma that concludes the proof of Theorem 4, we need to show the following lemma. Please note that this lemma has already been shown in slightly different form in [4], since the authors use the same idea in the Phase King algorithm.

▶ **Lemma 11.** *If a good node updates its variable* current *in Step* 6 *to x, no good node updates its variable* current *to y in Step* 6 *in the same phase, if $x \neq y$.*

Proof. Assume for a contradiction that there are two good nodes, one updating its value to x, and one updating its value to y. Therefore, one node must have heard at least $t + 1$ proposals for x, and another node heard at least $t + 1$ proposals for y. Thus, at least one good node proposed x, and one good node proposed y. In order for a good node to propose a certain value in Step 5, it must have heard this value at least $n - t$ times. Since at most t of those values can come from byzantine nodes, it must have heard the value from at least $n - 2t$ good nodes. Therefore at least $n - 2t$ good nodes sent x in Step 4, and at least $n - 2t$ good nodes sent y; with $x \neq y$, the total number of good nodes must be at least $2n - 4t$. Since the number of good nodes is by definition $n - t$, it follows that $n - t \geq 2n - 4t$, i.e., $n \leq 3t$ must hold, which contradicts our initial assumption that $n \geq 3t + 1$. ◀

With the help of this lemma, we only need to show the following lemma to complete the proof of Theorem 4.

▶ **Lemma 12.** *At the end of a phase with a non-byzantine jack, all good nodes store the same value in their variable* current*.*

Proof. If all good nodes store the same value in current at the beginning of the phase, this lemma follows directly from Lemma 6. Hence we only need to prove the case where not all good nodes start with the same value in current.

Note that the jack has two values which it can propose: Either the value stored in the suggestion created in the setup stage, or the value stored in its variable current. Both these values are valid: suggestion is chosen from the interval in the setup stage, and thus valid (see Lemma 3), and the value stored in current is always valid (see Corollary 10).

Let us first look at the case where the jack suggests its value stored in its variable current. As the jack suggests its current, it must have received more than t proposals for a certain value x, and therefore it has updated its current to x in Step 6. Since the jack heard more than t proposals, at least one good node proposed x. This node must have heard at x in Step 5 at least $n - t$ times, i.e., it heard it at least from $n - 2t$ good nodes. As we assume that $n \geq 3t + 1$, it follows that at least $t + 1$ good nodes started this phase with the value

x. As a consequence of Lemma 11, those nodes will not change the *value* in their variable current in Step 6, as the only value for which they can hear enough proposals is in fact x. This implies that the number of good nodes which have value x stored in current in Step 7 must be at least $t+1$, and because the jack suggests x, there will be at least $t+1$ good nodes that support x. Any good node that heard less than $n-t$ proposal messages for x will therefore accept the suggestion of the jack, which is x. Every node that ignores the suggestion from the jack does so, because it heard at least $n-t$ proposal messages for the same value x in Step 6. It follows from Lemma 11 that these nodes updated their current to the same value x as the jack did in Step 6. Therefore, all nodes have value x stored in current at the end of the phase.

It remains to be shown that if the jack suggests its value stored in suggestion, that all good nodes update their variable accordingly. If the jack suggests its suggestion, it heard at most t proposals for any value in Step 6. Hence, any other good node can hear at most $2t$ proposals for any value. Recall that the jack picked a value for its suggestion in the setup stage that was in at least $n-t$ bounds (note that there is always such a value, see Lemma 5). As at most t of those bounds might have been sent by byzantine nodes, the chosen value is in within at least $n-2t \geq t+1$ bounds of good nodes, where we used that $n \geq 3t+1$. Therefore, at least those $t+1$ good nodes support the suggested value in Step 7. Since every good node heard at most $2t < n-t$ proposals, and the suggestion of the jack has at least $t+1$ support, every good node accepts the suggested value of the jack in Step 8. ◀

Since we perform $t+1$ phases, we are guaranteed to have at least one phase with a non-byzantine jack. In combination with Corollary 7, this establishes the agreement property, and with Corollary 10 follows the median validity property. As the termination property follows from the construction of the algorithm, Theorem 4 follows. ◀

5 Discussion

In the following, we put our result into context, by providing a tight bound on how close to the median of the good nodes any algorithm can get, and by doing so, we also discuss an alternative approach to the initially stated problem of achieving agreement on a reasonable value.

▶ **Theorem 13.** *There is no deterministic algorithm that can guarantee that, for every selection of input values and every selection of byzantine nodes, the decision value x will always satisfy*

$$G[\left\lceil\frac{n-f}{2}\right\rceil - 1 - \left\lfloor\frac{t}{2}\right\rfloor] < x < G[\left\lceil\frac{n-f}{2}\right\rceil - 1 + \left\lceil\frac{t}{2}\right\rceil].$$

Proof. To prove this theorem, it is not necessary to assume byzantine *behavior* throughout the execution of the protocol; it suffices to let the byzantine nodes select tedious input values. We assume that $f = t$. Let us look at two different selections of input values: First, we let the start the good nodes with input values $G_1 = \{1, \ldots, n-t\}$, and the byzantine nodes pick the input values $\{n-t+1, \ldots, n\}$. Second, we start the good nodes with input values $G_2 = \{t+1, \ldots, n\}$, and the byzantine nodes with $\{1, \ldots, t\}$. Throughout the execution of the protocol, the byzantine nodes adhere to the protocol defined by the algorithm. As the two executions will be exactly equal, the good nodes agree on the same value v in both runs.

We will now show by contradiction that no value satisfies the bounds for both scenarios. Note that for all values which are in both sets of good input values G_1, G_2 it holds by construction that $G_1[i] = G_2[i-t]$. Assume that v is a value that satisfies the upper bound for G_1. Note that:

$$v < G_1[\left\lceil \frac{n-f}{2} \right\rceil - 1 + \left\lceil \frac{t}{2} \right\rceil] \implies v \leq G_1[\left\lceil \frac{n-f}{2} \right\rceil - 1 + \left\lceil \frac{t}{2} \right\rceil - 1]$$

Due to the selection of the input values of the good nodes. Therefore it must hold that:

$$v \leq G_1[\left\lceil \frac{n-f}{2} \right\rceil - 1 + \left\lceil \frac{t}{2} \right\rceil - 1] = G_2[\left\lceil \frac{n-f}{2} \right\rceil - 2 - \left\lfloor \frac{t}{2} \right\rfloor] < G_2[\left\lceil \frac{n-f}{2} \right\rceil - 1 - \left\lfloor \frac{t}{2} \right\rfloor]$$

Hence, every value that satisfies the upper bound for the first scenario, does not satisfy the lower bound for the second scenario. ◀

▶ **Theorem 14.** *There is a deterministic algorithm that guarantees that the decision value x always satisfies*

$$G[\left\lceil \frac{n-f}{2} \right\rceil - 2 - \left\lfloor \frac{t}{2} \right\rfloor] \leq x \leq G[\left\lceil \frac{n-f}{2} \right\rceil + \left\lceil \frac{t}{2} \right\rceil].$$

Proof. The algorithm works in two stages: First, we use any algorithm that achieves interactive consistency (for example the one from [24]). With interactive consistency, every node agrees on the input values of *all* nodes; in particular, it knows all the input values of the good nodes, and for all the byzantine nodes, there is either agreement on a value, or the good nodes agree that the node is byzantine. The second stage of the algorithm is to pick the median of the agreement array.

The agreement array will consist of all values from G, and some byzantine values. Note that every node that is identified as a byzantine node has no effect on the resulting decision value; therefore, the distortion created by the byzantine nodes is maximal if all byzantine nodes succeed in adding a bad value to the agreement array. Observe that this distortion can be maximized, if all byzantine nodes add values that are either smaller than the median of the good nodes, or larger[2]. It is clear that in order to achieve the maximal effect on the value of the resulting decision, and not only to maximize the index shift on the median of the good nodes, all byzantine values must be either smaller than the smallest good value, or larger than the largest good value, and that $f = t$. As $n \geq 3t + 1$, it is not possible for the byzantine nodes to move the median of the agreement array beyond the value range spanned by the good values; thus, the minimal (maximal) decision value can be bounded by a value from G. Due to the definition of the median, the t byzantine nodes can shift the decision value at most by $\lceil \frac{t}{2} \rceil$ index positions. Hence, the upper bound follows directly, and the lower bound follows with the fact that $\lceil \frac{t}{2} \rceil \leq \lfloor \frac{t}{2} \rfloor + 1$. ◀

Comparing how close to the median of the good nodes the decision value of the Jack algorithm is in terms of index position difference within G yields the following corollary:

▶ **Corollary 15.** *The Jack algorithm is a 2-approximation with respect to the index distance in G in the worst case.*

Having shown that the Jack algorithm is not optimal, one might argue, that it is better to use an algorithm based on interactive consistency, as outlined before. However, we are not

[2] If one byzantine node proposes a value that is smaller than the median, and one proposes a value that is larger, the median does not change.

aware of any algorithm that achieves interactive consistency with real valued[3] input values in an *efficient* way. The algorithm in [24] uses messages of exponential size, and also local computation that requires exponential runtime in each step. Since the Jack algorithm only requires small messages (only one[4] value), and only simple local computations, we claim that our algorithm achieves a reasonable trade-off between precision and complexity.

6 Conclusion

We introduced the median validity property which addresses a shortcoming in the specification of the byzantine agreement problem for many practical scenarios. Algorithms satisfying the median validity property can be used in environments with orderable input values, and such algorithms are particularly suitable if one cannot expect the input values of the nodes to be exactly equal. We presented an algorithm that achieves byzantine agreement satisfying this stronger validity property within an asymptotically optimal runtime of $t + 1$ phases, requiring only small messages. To analyze the quality of the validity property, we established a lower bound on the index distance to the good median within the input values of the good nodes, and we showed that our algorithm achieves a 2-approximation; i.e., if the input values are tightly clustered around a certain value v, the Jack algorithm guarantees to decide for a value that is very close to v.

References

1 Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

2 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

3 Piotr Berman and Juan A Garay. *Asymptotically optimal distributed consensus*. Springer, 1989.

4 Piotr Berman, Juan A Garay, and Kenneth J Perry. Towards optimal distributed consensus. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 410–415. IEEE, 1989.

5 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

6 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

7 Danny Dolev, Nancy A Lynch, Shlomit S Pinter, Eugene W Stark, and William E Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM (JACM)*, 33(3):499–516, 1986.

8 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM (JACM)*, 37(4):720–741, 1990.

9 Michael J Fischer and Nancy A Lynch. A lower bound for the time to assure interactive consistency. *Information processing letters*, 14(4):183–186, 1982.

[3] Note that many of the existing algorithms assume binary input values.

[4] In the setup stage two values are sent in one round, which could of course be split into two rounds in which only one value is sent.

10 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.

11 Matthias Fitzi and Juan A Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 211–220. ACM, 2003.

12 Oded Goldreich and Erez Petrank. The best of both worlds: Guaranteeing termination in fast randomized byzantine agreement protocols. *Information Processing Letters*, 36(1):45–49, 1990.

13 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

14 Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for byzantine fault detection. In *HotDep*, 2006.

15 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.

16 Soummya Kar and José MF Moura. Distributed consensus algorithms in sensor networks with imperfect communication: Link failures and channel noise. *Signal Processing, IEEE Transactions on*, 57(1):355–369, 2009.

17 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

18 Leslie Lamport and P Michael Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM (JACM)*, 32(1):52–78, 1985.

19 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

20 Jacob R Lorch, Atul Adya, William J Bolosky, Ronnie Chaiken, John R Douceur, and Jon Howell. The smart way to migrate replicated stateful services. *ACM SIGOPS Operating Systems Review*, 40(4):103–115, 2006.

21 Gil Neiger. Distributed consensus revisited. *Information Processing Letters*, 49(4):195–201, 1994.

22 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–320, 2014.

23 Stacy Patterson, Bassam Bamieh, and Amr El Abbadi. Distributed average consensus with stochastic communication failures. In *Decision and Control, 2007 46th IEEE Conference on*, pages 4215–4220. IEEE, 2007.

24 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

25 Wei Ren and Randal W Beard. *Distributed consensus in multi-vehicle cooperative control.* Springer, 2008.

26 Lili Su and Nitin Vaidya. Byzantine multi-agent optimization: Part I. *arXiv preprint arXiv:1506.04681*, 2015.

27 Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138. ACM, 2011.

28 Lin Xiao, Stephen Boyd, and Seung-Jean Kim. Distributed average consensus with least-mean-square deviation. *Journal of Parallel and Distributed Computing*, 67(1):33–46, 2007.

29 Lin Xiao, Stephen Boyd, and Sanjay Lall. A scheme for robust distributed sensor fusion based on average consensus. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 63–70. IEEE, 2005.

Ensuring Average Recovery with Adversarial Scheduler

Jingshu Chen[1], **Mohammad Roohitavaf**[2], **and Sandeep S. Kulkarni**[3]

1 **Michigan State University, East Lansing, USA**
`chenji15@cse.msu.edu`
2 **Michigan State University, East Lansing, USA**
`roohitav@cse.msu.edu`
3 **Michigan State University, East Lansing, USA**
`sandeep@cse.msu.edu`

Abstract

In this paper, we focus on revising a given program so that the average recovery time in the presence of an adversarial scheduler is bounded by a given threshold λ. Specifically, we consider the scenario where the fault (or other unexpected action) perturbs the program to a state that is outside its set of legitimate states. Starting from this state, the program executes its actions/transitions to recover to legitimate states. However, the adversarial scheduler can force the program to reach one illegitimate state that requires a longer recovery time.

To ensure that the average recovery time is less than λ, we need to remove certain transitions/behaviors. We show that achieving this average response time while removing minimum transitions is NP-hard. In other words, there is a tradeoff between the time taken to synthesize the program and the transitions preserved to reduce the average convergence time. We present six different heuristics and evaluate this tradeoff with case studies. Finally, we note that the average convergence time considered here requires formalization of hyperproperties. Hence, this work also demonstrates feasibility of adding (certain) hyperproperties to an existing program.

1998 ACM Subject Classification F.4 Mathematical logic and formal languages

Keywords and phrases Average Recovery Time, Hyper-liveness, Program Repair

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.23

1 Introduction

The problem of model repair focuses on revising a given program so that it satisfies new properties while preserving its existing properties. Such model repair is highly desirable when program requirements change (especially when new requirements are added) or bugs are identified in an existing program. The problem of model repair has been studied in the context of revising a program to add safety properties (e.g., to ensure that program never reaches an undesired state), liveness properties (e.g., if the program reaches a state where predicate X is true, then it will reach a state where some predicate Y is true), fault-tolerance properties (e.g., ensuring that safety and/or liveness is preserved in the presence of faults), and timing constraints [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15].

All of the properties considered in [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15] are expressed in terms of the framework in [2] that shows that any specification can be decomposed into a safety specification and a liveness specification. An important characteristic of the properties in [2] is that *Whether a program computation satisfies the specification is independent of other computations produced by that program.* So, if we consider a safety requirement of the

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 23; pp. 23:1–23:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

form: *the value of a variable x is never* 0 then we can evaluate a given program computation to decide whether x ever reaches 0. If yes, it implies that the computation violates the specification. It does not depend upon all other computations that the program can produce. Likewise, if the specification requires that: *if the program reaches a state where X is true then it will reach a state where predicate Y is true* and the given program computation satisfies this requirement then this observation remains true irrespective of other computations produced by that program. This implies that if we want to verify whether a given program is correct, we can evaluate each of its computations separately to determine if it satisfies the specification. If all of them satisfy the specification, we can identify that the program satisfies the specification. Otherwise, the program violates the specification.

Certain requirements however do not satisfy this constraint. Examples of this include some security properties (e.g., information flow [4], noninterference [17]) and some performance properties (e.g., average response time). To illustrate this, consider the requirement *if the program reaches a state where X is true then the average number of transitions required to reach a state in Y is* 5 *or less.* If a program has a computation where the response time (i.e., the number of steps required from X to Y) is 6 that does not imply that the specification is violated. In particular, if the program has several other computations with response time of 4 or less, then including the computation with response time of 6 is perfectly acceptable. In other words, properties such as average response time require analysis of all program computations simultaneously to decide whether program satisfies the specification or not. In [9], authors have introduced the notion of hyperproperties to characterize such requirements. They have also shown that hyperproperties are strictly more general than the (simpler) properties identified in [2].

Existing work in [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15] is designed for addition of properties from [2] and does not address performing model repair to add hyperproperties. With this motivation, in this paper, we focus on developing complexity results and algorithms for the addition of one type of hyperproperty, namely average response time.

To motivate the requirement considered in this paper, consider a typical requirement in the context of fault-tolerant and/or stabilizing programs: In these programs, it is required that after faults stop occurring, the program recovers to a legitimate state. An important attribute for this recovery is the time taken for it. There are several ways –worst case, average case etc – to compute the recovery time. The recovery time is also affected by assumptions made about any non-deterministic choices the program may face. In our work, we consider the following approach to compute the average time for recovery (convergence). We focus on the case where the fault perturbs the program to an illegitimate state and the program recovers from there to a legitimate state. Since faults are typically random in nature, there is a probability distribution associated with illegitimate states. (For sake of simplicity, in our case studies, we assume that all illegitimate states are reached equally likely. However, our approach can handle any probability distribution.) Subsequently, during program recovery, there is often a non-deterministic choice given to the program. When faced with such a choice, we consider the case where we use adversarial scheduler that attempts to force the program down on a path that increases the convergence time. This enables one to account for an implementation where the designer considers the non-deterministic choices in any arbitrary order.

Based on the choices considered above, we focus on ensuring that the average recovery time in the presence of an adversarial scheduler (denoted as average convergence time for brevity) is less than λ_e. Furthermore, during this repair, we want to preserve existing safety and liveness properties. Hence, during repair, we focus on removing existing behaviors so that the average convergence time is less than λ_e.

Contributions of the paper. The main contributions of the paper are as follows:

- Since repair for satisfying the average response time constraint requires removal of behaviors/transitions that are responsible for increasing the convergence time, we evaluate the complexity of revision for satisfying the average convergence time requirement while removing a minimum number of transitions. We show that this problem is NP-hard.
- We also show that if we omit the requirement about removing only a minimum transitions then the problem can be solved in P. We present an approach (denoted as SCP) to evaluate this approach. While it is very fast, we find that it removes a large number of transitions (in some cases $> 99\%$).
- To overcome the limitations of SCP and the NP-hardness of maximizing the number of transitions, we present five additional heuristics namely ELP, KBP, RIA, RIAD and SSP. Of these, RIA and RIAD take into account possible distribution constraints that prevent a process from reading or writing all program variables. We show that these approaches provide a tradeoff between the time required to find the desired program and the number of transitions that are removed to guarantee average convergence time.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we define the notion of programs and average convergence time. In Section 3, we define the problem of adding average convergence time. The complexity analysis of this problem is discussed in Section 4. In Section 5, we present our six approaches that identify the tradeoff between the time for repair and the non-determinism preserved in the repaired program. We discuss our experimental results for two case studies in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Preliminaries

In this section, we formally introduce the notions of program and other related definitions. Our definitions are based on those given by Arora and Gouda [3].

▶ **Definition 1** (Program). A program $\mathcal{P}$ is a tuple $\langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, where:

- $S_{\mathcal{P}}$ is the (finite) state space, i.e., the set of all states of $\mathcal{P}$.
- $\delta_{\mathcal{P}}$ is a set of transitions. Specifically, $\delta_{\mathcal{P}} \subseteq S_{\mathcal{P}} \times S_{\mathcal{P}}$.

For simplicity of presentation, we assume that there is at least one outgoing transition of $\mathcal{P}$ from each state. If there is no transition from state s, we can simply add transition (s, s). We would like to note that this is not a restriction in any sense. However, it avoids having to consider terminating states in subsequent definitions.

▶ **Definition 2** (State Predicate). Given a program $\mathcal{P}$ $(\langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle)$, a state predicate $\mathcal{P}$ is a subset of $S_{\mathcal{P}}$.

▶ **Definition 3** (Computation). A computation of $\mathcal{P}$ is an infinite sequence of states, $\rho = \langle s_0, s_1, \dots \rangle$, where

- $\forall j, j > 0$: $(s_{j-1}, s_j) \in \delta_{\mathcal{P}}$.

▶ **Definition 4** (Distance of a state predicate in a computation). Let $\rho = \langle s_0, s_1, \dots \rangle$ be a computation of $\mathcal{P}$. Let S be a state predicate of $\mathcal{P}$. We say that the distance of ρ to S (denoted by $compdist(\mathcal{P}, \rho, S)$) is w iff $\forall j : (j < w) \Rightarrow s_j \notin S$ and $s_w \in S$.

In the above definition, if ρ does not contain a state in S, we say that $compdist(\mathcal{P}, \rho, S) = \infty$. Next, we overload the definition of distance to define the notion of a distance of a state predicate from a given state, say s. There may be several computations that start from s. Since we focus on an adversarial scheduler, distance of a state s from state predicate S is described by considering the maximum number of steps required from s in some computation of $\mathcal{P}$. In other words, this definition captures the maximum distance from state s to state predicate S.

▶ **Definition 5** (Distance of a state predicate from a state)**.** Distance of state s to a state predicate S in program $\mathcal{P}$, denoted by $statedist(\mathcal{P}, s, S)$, is $max(compdist(\mathcal{P}, \rho, S) | \rho$ is a computation of $\mathcal{P}$ that starts in $s)$.

Using the above definition, we can define the notion of average time to recover from some state predicate T to another state predicate S as follows:

▶ **Definition 6** (Distance between two state predicates)**.** Let S and T be state predicates of $\mathcal{P}$. The average convergence time from T to S in program $\mathcal{P}$, denoted by $predist(\mathcal{P}, T, S)$ is $average(statedist(\mathcal{P}, s, S) | s \in T - S)$.

For sake of simplicity, we define $predist(\mathcal{P}, S, S)$ to be 0.

▶ **Definition 7** (Average convergence time)**.** Let S and T be state predicates of $\mathcal{P}$. Let λ be a real number. We say that T converges to S within λ iff

- $predist(\mathcal{P}, T, S) \leq \lambda$.

Observe that if some computation of $\mathcal{P}$ starts from a state in T and never reaches a state in S then $predist(\mathcal{P}, T, S) \geq \lambda$ from any number λ.

3 Problem Formulation

In this section, we formally state our program repair problem with respect to average convergence time requirements. The goal of this problem is to revise a given program $\mathcal{P}$ to $\mathcal{P}'$ that uses a subset of behaviors of $\mathcal{P}$ to reduce the convergence time to the set of legitimate states. Since $\mathcal{P}'$ only uses a subset of behaviors of $\mathcal{P}$, it follows that if $\mathcal{P}$ satisfied any safety or liveness property (that is described using the framework [2]) then $\mathcal{P}'$ satisfies that property as well.

The input to the repair program consists of program $\mathcal{P}$ with state space $S_\mathcal{P}$ and transitions $\delta_\mathcal{P}$. It also includes the state predicate denoting the legitimate states, S. Finally, it includes the desired average convergence time λ. The goal of the program is to identify program $\mathcal{P}'$ that uses the behaviors of $\mathcal{P}$, and provides convergence to S with average time λ. Thus, the problem statement is as follows:

▶ **Definition 8** (The Program Repair Problem)**.** Given a program $\mathcal{P} = \langle S_\mathcal{P}, \delta_\mathcal{P} \rangle$, the set of legitimate states S, and the required average convergence time λ, identify $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$ such that

- $S_{\mathcal{P}'} \subseteq S_\mathcal{P}$
- $\delta_{\mathcal{P}'} \subseteq \delta_\mathcal{P}$
- $\mathcal{P}'$ converges to S from $S_{\mathcal{P}'}$ within λ', $\lambda' \leq \lambda$.

In order to characterize the complexity of the above problem, we identify a corresponding decision problem. The first attempt to find this decision problem is as follows:

▶ **Definition 9** (The Decision Problem (Attempt 1) (Dec_1).)**.** Given a program $\mathcal{P} = \langle S_\mathcal{P}, \delta_\mathcal{P} \rangle$, the set of legitimate states S, and the required average convergence time λ: Does there exist a program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$ that satisfies the requirements specified for the program repair problem in Definition 8.

The decision problem Dec_1 can be trivially answered by setting $S_{\mathcal{P}'}$ to S. In this case, it is straightforward that $\mathcal{P}$ converges to S from S within 0. To avoid such trivial answer, we require that recovery from all states in $S_{\mathcal{P}}$ be preserved. Hence, we require that $S_{\mathcal{P}'} = S_{\mathcal{P}}$. Hence, the second attempt at defining the decision problem is as follows:

▶ **Definition 10** (The Decision Problem (Attempt 2) (Dec_2))**.** Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, the set of legitimate states S, and the required average convergence time λ: Does there exist a program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$, such that $S_{\mathcal{P}'} = S_{\mathcal{P}}$ and $\mathcal{P}'$ satisfies the requirements specified for the program repair problem in Definition 8.

The decision problem Dec_2 can also be solved in P (in the state space of the program) as follows: We start with $\mathcal{P}'$ that contains no transitions in $S_{\mathcal{P}} - S$. Then, from each state in $S_{\mathcal{P}} - S$, we add the shortest path (least recovery in terms of number of transitions) to some state in S. If there are several shortest paths, we can choose any one of them. We argue (in Section 4) that the resulting program guarantees that starting from any state in $S_{\mathcal{P}}$, the program reaches a state in S. Also, the resulting program provides the least average convergence time. Hence, if the average convergence time is larger than λ then it is impossible to find $\mathcal{P}'$ that satisfies the problem statement Dec_2.

In both decision problems Dec_1 and Dec_2, we require that $\delta_{\mathcal{P}'} \subseteq \delta_{\mathcal{P}}$. Requiring $\delta_{\mathcal{P}'} = \delta_{\mathcal{P}}$ is meaningless since it would require $\mathcal{P}$ and $\mathcal{P}'$ to be identical. However, we can focus on finding $\mathcal{P}'$ that preserves the maximum behavior of $\mathcal{P}$. Having $\mathcal{P}'$ with more non-determinism is desirable, as it provides the designer with a maximum choice in terms of implementation. It is also known to increase the ability to add new properties in the future. Thus, we define the decision problem as follows:

▶ **Definition 11** (The Decision Problem (Final) (Dec_3))**.** Given a program $\mathcal{P} = \langle S_{\mathcal{P}}, \delta_{\mathcal{P}} \rangle$, the set of legitimate states S, the required average convergence time λ, **and integer** k: Does there exist a program $\mathcal{P}' = \langle S_{\mathcal{P}'}, \delta_{\mathcal{P}'} \rangle$, such that $S_{\mathcal{P}'} = S_{\mathcal{P}}$, $|\delta_{\mathcal{P}'}| \geq k$, and $\mathcal{P}'$ satisfies the requirements specified for the program repair problem in Definition 8.

In Section 4, we show that Dec_3 is NP-complete in the state space $S_{\mathcal{P}}$. Given that Dec_2 is in P but Dec_3 is NP-complete, it follows that there is a tradeoff between the time required to find $\mathcal{P}'$ and the fraction of transitions/behaviors removed by $\mathcal{P}'$. In particular, it is efficient to find $\mathcal{P}'$ that preserves only a small subset of behaviors. However, it is significantly more complex to design $\mathcal{P}'$ that designs the maximum possible behaviors.

4 Complexity Analysis

In this section, we show that Dec_2 can be solved in polynomial time in the state space of the program, using a straightforward approach, but Dec_3 is NP-complete.

Regarding Dec_2, we construct transitions of $\mathcal{P}_{minpath}$ as follows: For each state, $s \notin S$, we include the transitions corresponding to the path L_s which is the shortest path from s to some state in S. Next, in Lemmas 12 and 13, we show that the resulting program provides the least average convergence time for any program that solves the problem in Definition 8. Hence, if this program does not provide the desired average recovery time then the answer to Dec_2 is false.

▶ **Lemma 12.** *$\mathcal{P}_{minpath}$ guarantees that starting from any state in S_p, the program reaches a state in S.*

▶ **Lemma 13.** *$\mathcal{P}_{minpath}$ provides the least average recovery time for any program that solves the problem in Definition 8.*

▶ **Theorem 14.** *Dec_2 can be solved in polynomial time in the state space of the input program $\mathcal{P}$.*

Regarding Dec_3, we can reduce the problem of adding liveness constraints in [5] to Dec_3. Specifically, in [5], it is shown that the following problem is NP-complete.

Given a program $\mathcal{P}$, two state predicates S and T and a positive integer k, does there exist a $\mathcal{P}'$ such that $S_{\mathcal{P}'} = S_{\mathcal{P}}$, $\delta_{\mathcal{P}'} \subseteq \delta_{\mathcal{P}}$, every computation of $\mathcal{P}'$ that starts in a state predicate T reaches a state in state predicate S and $\delta_{\mathcal{P}'} \geq k$.

Showing that Dec_3 is in NP is straightforward. To reduce the above problem to an instance of Dec_3, we essentially need to set the value of λ, the average convergence time of $\mathcal{P}'$ to $|S_{\mathcal{P}}|$. It is straightforward to observe that if every computation of $\mathcal{P}'$ reaches the state predicate S then the average convergence time is less than $|S_{\mathcal{P}}|$. Thus, we have

▶ **Theorem 15.** *Dec_3 is NP-complete in the state space of the input program $\mathcal{P}$.*

5 Repair Approaches

In this section, we consider the problem of repairing a given program $\mathcal{P}$ to meet the average convergence time λ requirement. As shown in Theorem 15, this problem is NP-hard under the constraint that the revised program must preserve a given number of transitions. By contrast, if we solve this problem minimally (Dec_2) without the above constraint, then the problem can be solved in P (cf. Theorem 14). However, the solution for this approach (discussed in Section 5.1) is likely to include only a small number of transitions in the repaired program. In other words, there is a tradeoff between the time required to design the repaired program and the level of non-determinism (choices) available to that repaired program. Hence, we evaluate this tradeoff with several heuristics. At one extreme, we consider the approach that is expected to be the fastest but provides least non-determinism. This approach is based on the algorithm that solves Dec_2. The other extreme, i.e., the solution with maximum choices, requires exponential time (unless $P = NP$) and, hence, is infeasible. We also consider several intermediate heuristics as well.

We develop the following six approaches. Of these, the first approach, Shortest Convergence Path (SCP), focuses on adding those transitions which lead to the shortest convergence paths. The second approach, Eliminate Longest Path (ELP), focuses on removing offending behaviors that cause an increase in the delay of convergence. The third approach, Keep Best Path (KBP), repairs the given program by only preserving transitions that lead to the shortest convergence paths when several outgoing transitions are available for states.

These first three approaches view the program purely in terms of its transitions. They ignore the structure of the program. In the next two heuristics, we focus on the structure of the program in terms of guarded commands [11]. Specifically, the forth approach, Restrict Individual Actions (RIA), uses the guarded commands of the input program $\mathcal{P}$ and constructs $\mathcal{P}'$ whose guards are a combination of guards involved in $\mathcal{P}$. The fifth approach, Restrict Individual Actions with Distribution Restrictions (RIAD), extends RIA to deal with restrictions imposed by distributed systems. In particular, it restricts the actions whose guards can be combined. This allows one to ensure that the repaired program can be implemented in low atomicity where each process can read or write only a subset of variables. Hence, RIAD provides a mechanism to guarantee average convergence time to distributed systems where each process can read/write only a subset of program variables.

Finally, the sixth approach, Solve Similar Problem (SSP) partitions the problem into two subproblems. Of these, in the first step, we focus on guaranteeing worst case convergence time with value that is larger than the desired average convergence time. And, in the second step, we apply ELP on the resulting program.

In all these approaches, our algorithm takes as input the program $\mathcal{P}$, its set of legitimate states S and the desired average convergence time λ_e. The algorithm returns the desired program $\mathcal{P}'$ if a solution is found that solves the problem in Definition 8. Otherwise, it returns ϕ.

5.1 Approach 1 (SCP): A Refinement Procedure via Including Shortest Convergence Paths

In this section, we present SCP (Shortest Convergence Path) – a fast heuristic that focuses on reducing convergence time without considering the number of preserved transitions. Based on the idea of Dec_2, SCP repairs a given program by preserving only those transitions that lead a program to shortest convergence paths. This reduces/eliminates choices that the scheduler can play. However, it is anticipated that it would eliminate a large number of transitions/behaviors. To identify transitions that lead to the shortest convergence path, our approach SCP performs a backward computation from legitimate states.

Figure 1 gives pseudocode for the overall refinement algorithm of SCP. We initialize the revised program to $\emptyset$ and set the current reachable state set to the set of legitimate states. In each iteration of the RepairBySCP loop, starting from current reachable states, we perform a backward computation to identify transitions that lead to the shortest convergence path. Based on such one-step backward search per iteration, we identify newly reachable state S_{next} (Line 8) and calculate the transitions that lead program from $S_{next}, S_{current}$ (Line 9). Then, we update the current reachable state set in Line 10. In this step, our implementation simply performs $S_{current} \cup S_{next}$. Now, we use the transitions computed in Line 9 to update the current revised program (Line 11). In particular, our implementation simply performs $\delta_{current} \cup \delta_{tmp}$. Based on the updated program transitions, we re-calculate the average convergence time of $\mathcal{P}^c$. There are two scenarios for our algorithm to stop the *while* loop. One is that if we reach a program $\mathcal{P}^c$ whose average convergence time is larger than λ_e, the loop will stop. As in Line 7, $\mathcal{P}'$ records the revised program that fits the average convergence time requirement. The other one is that our refinement procedure has included all the transitions in the shortest convergence path and the generated program fits the average convergence time requirement, that is, $\lambda_r < \lambda_e$. In this case, our refinement process will break the loop.

Hence, after executing such an iterative refinement procedure, our refinement algorithm generates a program $\mathcal{P}'$ that holds the maximum $\lambda_{\mathcal{P}'}$ and $\lambda_{\mathcal{P}'} \leq \lambda_e$.

5.2 Approach 2 (ELP): A Refinement Procedure via Eliminating Maximal Transitions

In this section, we present ELP (Eliminate Maximal Path) –a heuristic that focuses on reducing convergence time while preserving maximum non-determinism.

The key idea of ELP is to find the set S_{max}, the set of states from where the (worst case) convergence path is the longest. Let λ_{worst} be this worst case convergence path. Let S_{next} be the set of states from where the worst case convergence path is $\lambda_{worst} - 1$. After finding S_{max} and S_{next}, we remove transitions $\{(s_1, s_2) | s_1 \in S_{max} \wedge s_2 \in S_{next}\}$. This process is repeated until we find the desired program or conclude that realizing such a program is impossible. As an illustration, consider the Figure 3. This figure shows four states s_1, s_2, s_3 and s_4. Assume that the worst case convergence path from s_2, s_3 and s_4 are $10, 7$ and 5

```
RepairBySCP($\mathcal{P}, \lambda_e$):
Input     $\lambda_e$: the expected average convergence time.
          $\mathcal{P}$: transitions $\delta_{\mathcal{P}}$ and invariant $S_{\mathcal{P}}$.
Output    $\mathcal{P}'$: $\lambda_{\mathcal{P}'} \leq \lambda_e$
    $S_{current} = invariant$;
    $\delta_{current} = empty$;
    $\mathcal{P}' = \emptyset$;
    $\mathcal{P}^c = \emptyset$;
    do
    {
        $\mathcal{P}' = \mathcal{P}^c$;
        $S_{next} \leftarrow BackwardOneStepCompute(S_{current});$
        $\delta_{tmp} \leftarrow CalculateTransition(S_{next}, S_{current});$
        $S_{current} \leftarrow UpdateCurrentState(S_{current}, S_{next});$
        $\mathcal{P}^c \leftarrow RefineProgramTransitions(\delta_{current}, \delta_{tmp});$
        $\lambda_r \leftarrow CalculateAvgConvTime(\mathcal{P}^c);$
        if  $(\mathcal{P}^c = \emptyset)$
            break ;
    }
    while $(\lambda_r > \lambda_e)$;// $\lambda_r$ is the current average convergence time after refinement.
    Return $\mathcal{P}'$;
```

Figure 1 SCP: program repair by preserving shortest convergence path.

```
RepairByELP($\mathcal{P}, \lambda_e$):
Input     $\lambda_e$: the expected average convergence time.
          $\mathcal{P}$: transitions $\delta_{\mathcal{P}}$ and invariant $S_{\mathcal{P}}$.
Output    $\mathcal{P}'$: $\lambda_{\mathcal{P}'} \leq \lambda_e$
1       $\mathcal{P}' = \mathcal{P}$;
2       do
3       {
4           $S_{max} \leftarrow CalculateMaximalState(\mathcal{P}');$
5           $S_{next} \leftarrow CalculateNextMaxState(\mathcal{P}');$
6           $\delta_{maxGroup} \leftarrow CalculateMaxTrans(S_{max}, S_{next});$
7           $\mathcal{P}' \leftarrow RefineProgramTransitions(\mathcal{P}^c, \delta_{maxGroup});$
8           $\lambda_r \leftarrow CalculateAvgConvTime(\mathcal{P}^c);$
9           if  $(\mathcal{P}' = \emptyset)$
10              break ;
11      }
12      while $(\lambda_r > \lambda_e)$;// $\lambda_r$ is the current average convergence time after refinement.
13      Return $\mathcal{P}'$
```

Figure 2 ELP: A Refinement Procedure via Eliminating Maximal Transitions.

respectively. In that case, worst case path from s_1 is 11. Also, assume that $s_1 \in S_{max}$. In this case, we remove the transition (s_1, s_2).

Figure 2 gives the pseudo code for the overall refinement algorithm. The procedure RepairByELP repairs the given program $\mathcal{P}$ top-down, starting with original program transition $\delta_{\mathcal{P}}$. In each iteration, we calculate the maximal state set S_{max} (Line 4) and the next-maximal state set S_{next} (Line 5) for the current revised program $\mathcal{P}'$. With S_{max} and S_{next}, we calculate a group of maximal transitions. As in Line 6, we calculate maximal transitions (denoted as $\delta_{maxGroup}$). Then in Line 7, we repair program by eliminating $\delta_{maxGroup}$ from current program transitions set. Line 8 calculates the current average (maximal) convergence time for $\mathcal{P}'$. Line 9 describes a possible case where our algorithm reaches an empty program. If this case occurs, Line 10 would break the computation loop. Otherwise, the whole RepairByELP loop will stop when it reaches a point where current average convergence time is less than λ_e. The resulting program $\mathcal{P}'$ fits the average convergence time requirement, that is $\lambda_{\mathcal{P}'} \leq \lambda_e$.

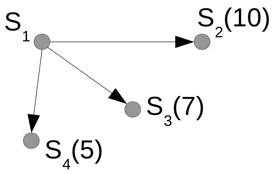


Figure 3 Four states s_1, s_2, s_3 and s_4.

```
RepairByKBP($\mathcal{P}, \lambda_e$):
Input    $\lambda_e$: the expected average convergence time.
         $\mathcal{P}$: transitions $\delta_{\mathcal{P}}$ and invariant $S_{\mathcal{P}}$.
Output   $\mathcal{P}'$: $\lambda_{\mathcal{P}'} \leq \lambda_e$
1      $\mathcal{P}' = \mathcal{P}$;
2      do
3      {
4          $S_{max} \leftarrow$ CalculateMaxState($\mathcal{P}'$);
5          $\delta_{nonMin} \leftarrow$ CalculateNotMinTrans($S_{max}$);
6          $\mathcal{P}' \leftarrow$ RefineProgram($\mathcal{P}', \delta_{nonMin}$);
7          $\lambda_r \leftarrow$ CalculateAvgConvTime($\mathcal{P}'$);
8          if ($\mathcal{P}^c = \emptyset$)
9              break ;
10     }
11     while ($\lambda_r > \lambda_e$);// $\lambda_r$ is the current average convergence time after refinement.
12     Return $\mathcal{P}'$;
```

Figure 4 KBP: A Refinement Procedure via Removing NonMinimum Transitions.

5.3 Approach 3 (KBP): A Refinement Procedure via Eliminating NonMinimum Transitions

In this section, we present KBP (Keep Best Path) – a heuristic that focuses on reducing convergence time with considering preservation of least non-determinism for those maximal states. Similar to the approach ELP, during the refinement procedure, our approach KBP iteratively removes a group of transitions from current program transition set until we reach a point where the revised program fits the average convergence time requirement. The difference between KBP and ELP is that we remove not only the maximal transitions from S_{max} but also other transitions except those that provide the best recovery time.

Once again, consider the transitions in Figure 3. In this figure, assuming that $s_1 \in S_{max}$, we keep the transition (s_1, s_4) and remove (s_1, s_2) and (s_1, s_3). Observe that in this case, we are removing more transitions while making a bigger impact on the average convergence time. Compared with ELP, we expect that KBP will reduce the time for repair but it will result in more transitions being removed.

The procedure RepairByKBP in Figure 4 repairs the given program top-down, starting with original transitions. Specifically, we calculate S_{max} in Line 4 and corresponding nonminimum transition set δ_{nonmin} for each state in S_{max}. Then, we repair program by eliminating δ_{nonmin} from current program transitions (Line 6). Line 7 calculates the current average (maximal) convergence time for $\mathcal{P}'$. Line 8 describes a possible case where our algorithm reaches an empty program. If this case occurs, Line 9 would break the computation loop. The resulting program $\mathcal{P}'$ is one solution that fits the average convergence time requirement, that is, $\lambda_{\mathcal{P}'} \leq \lambda_e$.

5.4 Approach 4 (RIA): A Refinement Procedure via Revising Maximal *Actions* with Minimal *Actions*

While the previous three approaches focused on repair at transition level, in this approach, we focus on additional structure in the given program to perform the repair. This allows one to take into consideration problems that arise in distributed systems as well as possible limitations on how programs are evaluated.

Before we describe our approach, we consider the case where the state space is more compactly represented by variables and program transitions are compactly represented using guarded commands of the form $g \longrightarrow st$. In particular, in this case, the state space is obtained by assigning each variable value from its respective domain. And, transitions corresponding to an action $g \longrightarrow st$, where g is a Boolean expression involving program variables and st is a statement that updates those program variables, are represented by the set $\{(s_0, s_1)|g$ evaluates to true in s_0 and s_1 is obtained by updating those variables as prescribed by $s_1\}$ [1]

Specifically, in this approach, we focus on revising the given program so that the guards and statements in the repaired program are *comparable* to that in the original program. To illustrate our approach, consider Figure 3. In this figure, let the transition (s_1, s_a) be executed by the action $g_a \longrightarrow st_a$, where $2 \leq a \leq 4$. In this figure, in approach ELP, we removed the transition (s_1, s_2). In RIA, we achieve the same by restricting the corresponding action $g_2 \longrightarrow st_2$ to be executed only if the action corresponding to (s_1, s_4) is not enabled. In other words, we change the action to $g_2 \wedge (\neg g_4) \longrightarrow st_2$. Observe that this change causes removal of additional transitions that start from a state where g_2 is true and g_4 is false. This approach is based on the heuristic that this overall change will result in reduction in the average convergence time. Observe that with this change, the guards involved in the repaired program are a combination of the guards involved in the original program. And, the statements in the repaired program are same as that in the original program. Since the guards of the original program represent predicates that could be checked in the original program, this allows the user to control the types of actions that can appear in the repaired program.

Figure 5 gives the pseudo code for the overall refinement algorithm. Specifically, Line 4 computes S_{max}, the state set in which each state has a possibility to reach maximal convergence path. Line 5 calculate the maximal actions for each state in S_{max}. Line 6 calculate the minimal actions for each state in S_{max}. Line 7 revise the maximal actions for each state in S_{max} using the corresponding minimal actions. Then Line 8 refines program by repairing those maximal actions from current program transition set. Line 9 calculates the current average (maximal) convergence time for $\mathcal{P}'$. Line 10 describes a possible case where our algorithm reaches an empty program. If this situation occurs, Line 11 would break the computation loop. Otherwise, the resulting program $\mathcal{P}'$ satisfies the average convergence time, that is $\lambda_{\mathcal{P}'} \leq \lambda_e$.

After executing such an interatively refinement procedure, our refinement algorithm either reaches an empty program or returns a program that fits the average convergence time requirement.

[1] As an illustration, if the program had two variables x and y with domain $\{0, 1\}$ and $\{0, 1, 2\}$ respectively then the state space contains 6 possible states $00, 01, 02, 10, 11$ and 12 where the first value denotes the value of x and the second denotes the value of y. And, action $x = y \longrightarrow x = 0$ corresponds to transitions $(00, 00), (11, 01)$.

```
RepairByRIA($\mathcal{P}, \lambda_e$):
Input    $\lambda_e$: the expected average convergence time.
         $\mathcal{P}$: transitions $\delta_{\mathcal{P}}$ and invariant $S_{\mathcal{P}}$.
Output   $\mathcal{P}'$: $\lambda_{\mathcal{P}'} \leq \lambda_e$
    $\mathcal{P}' = \mathcal{P}$;
    do
    {
        $S_{max} \leftarrow$ CalculateMaxState($\mathcal{P}'$);
        $act_{max} \leftarrow$ CalculateMaxAct($S_{max}, \mathcal{P}'$);
        $act_{min} \leftarrow$ CalculateMinAct($S_{min}, \mathcal{P}'$);
        $act'_{max} \leftarrow$ RepairActions($act_{max}, act_{min}$);
        $\mathcal{P}' \leftarrow$ RefineProgram($act'_{max}, \mathcal{P}'$);
        $\lambda_r \leftarrow$ CalculateAvgConTime($\mathcal{P}'$);
        if ($\mathcal{P}' = \emptyset$)
            break ;
    }
    while ($\lambda_r > \lambda_e$);// $\lambda_r$ is the actual average convergence time of $\mathcal{P}$.
    Return $\mathcal{P}'$;
```

Figure 5 RIA: A Refinement Procedure via Revising Maximal *Actions* with Minimal *Actions*.

5.5 Approach 5 (RIAD): A Refinement Procedure via Revising Maximal Actions with Distribution Consideration

In this section, we extend RIA to take into account what guards could be used in repairing a given action. In turn, this allows to fully capture the requirements of a distributed system. To illustrate this, consider the case where the nature of distributed systems prevents a process from accessing all program variables. Rather, each process is only allowed to read and write a subset of variables.

Recall that in RIA, we restricted the guard of one action by negation of the guard of another action. Given a guard, it is straightforward to identify the variables that it is allowed to read. Hence, for each action, we identify *neighborhood* actions that can be used to restrict it while preserving the read/write restrictions. By only selecting this subset of actions, we can ensure that the synthesized program satisfies the read/write restrictions of the given system. Since RIAD is similar to RIA (except for this neighborhood restriction), we do not provide a detailed algorithm for RIAD.

5.6 Approach 6 (SSP): A Refinement Procedure via Eliminating Maximal Transitions from a Reduced Program

In this section, we propose SSP (Solve Simpler Problem). The main idea of this algorithm is to use the same repair technique as ELP (from Section 5.2) on a program $\mathcal{P}_r$ (described next) rather than the original program $\mathcal{P}$.

Let λ_e be the desired average convergence time. Let λ_{worst} be the worst case convergence time of $\mathcal{P}$. The goal of $\mathcal{P}_r$ is to ensure that the worst case convergence time is bounded by $\lambda_{\mathcal{P}_r}$ where $\lambda_{worst} \geq \lambda_{\mathcal{P}_r} \geq \lambda_e$. Subsequently, we utilize ELP to remove any behaviors from $\mathcal{P}_r$ to ensure that the average convergence requirements are satisfied.

The motivation behind SSP is that each step involved could be implemented efficiently. Specifically, the first step, which involves ensuring worst case behavior, is simpler. This is due to the fact that worst case analysis of repaired programs is substantially easier than average case behavior. Also, the transitions removed in the first step are good candidates for removal from the desired program $\mathcal{P}'$. Also, it is anticipated that $\mathcal{P}_r$ is close to the desired program $\mathcal{P}'$. Hence, the amount of time involved in the second step would be small as well.

Observe that SSP provides a continuum of possible values for $\lambda_{\mathcal{P}_r}$. At one extreme, choosing $\lambda_{\mathcal{P}_r} = \lambda_{worst}$ will result in SSP to be equivalent to ELP. At another extreme, choosing $\lambda_{\mathcal{P}_r} = \lambda$ will result in unnecessary removal of transitions in the first step and obviate the need for the second step. For the sake of analysis, we choose $\lambda_{\mathcal{P}_r}$ to be the average of λ_{worst} and λ_e. Since the construction of $\mathcal{P}_r$ is straightforward and the remaining algorithm is same as ELP, we do not provide detailed algorithm for SSP.

6 Case Study & Experiment Results

We have developed a tool $\mathcal{R}time$ that implements the six approaches described previously. $\mathcal{R}time$ takes as input the following parameters:

- the input program $\mathcal{P}$,
- the set of legitimate states, S
- the desired convergence time, λ, and
- approach to be used for adding average convergence time

It identifies the program that satisfies the requirements of Problem 8. For the sake of analysis, we allow $\mathcal{R}time$ to output a program even if it removes all transitions from some state $s \in S_\mathcal{P}$. When this happens, the fraction of transitions that are preserved will be lower as well. Since our goal is to compare the level of non-determinism left in the program and the time taken for synthesis, this allows us to compare the different approaches directly.

Observe that all our approaches are sound by construction, i.e., when they output a program, we have already validated that the average convergence time of that program is less than the given parameter λ. Also, since these programs only use polynomial time, the number of transitions they preserve is not necessarily maximum. Also, if any of these approaches remove all transitions from some state s, making s be a deadlock state then they cannot satisfy $S_{\mathcal{P}'} = S_\mathcal{P}$. However, instead of declaring failure in this case, we report the number of transitions still preserved in the program. This allows us to compare all approaches in all examples. Note that the worst case is that all states outside S become deadlock states. In this case, the fraction of preserved transitions will be 0.

We now demonstrate our approaches on a classic stabilizing algorithm, which is K-state token ring program [10] and the Stabilizing Tree based algorithm [21] that is obtained adding stabilization to the classic mutual exclusion algorithm by Raymond [21]. All the experiments are performed on an Intel Core i7 machine 2.90GHz with 8GB memory. Also, the reachability analysis required for the different approaches is performed with the BDD package [8].

6.1 K-state Token Ring Program

We give a brief description of the K-state token ring program from [10]. The program $\mathcal{P}_{tk}$ consists of n processes, $0..(n-1)$, that are arranged in a unidirectional ring. For each process p_i, it has one variable x_i with domain $\{0,\ 1, \ldots \text{K-1}\}$.

$$\begin{array}{llcl} Action_0: & x_0 == x_{n-1} & \longrightarrow & x_0 = (x_0+1) \bmod K; \\ Action_1: & x_i \neq x_{i-1} & \longrightarrow & x_i = x_{i-1}; \end{array}$$

In the above two actions, $Action_0$ is only for process p_0. When $x_0 == x_{n-1}$ is satisfied, this action is enabled for execution. If chosen for execution, process 0 increments x_0 by 1 in modulo K arithmetic. $Action_1$ is for all other processes p_i, $i \neq 0$. When x_i differs from x_{i-1}, $Action_1$ is enabled for execution. When p_i executes its action, it sets x_i to the value of x_{i-1}.

Legitimate states. The legitimate states of the program are those states where only one token is circulated along the ring. To calculate this set, we start from a state where all x values are 0. Then, we compute all the states reached by the execution of the above program.

▶ Remark. In subsequent analysis, we let $K = n$ to ensure that convergence to legitimate states is guaranteed.

We conduct our experiments for repairing the token-ring program $\mathcal{P}_{tk}$ with different average convergence requirements. Instead of using specific real number values for the desired average convergence time, we use a fraction of the existing worst case convergence time. This is due to the fact that the time required to obtain an average convergence time of 10 for 3 processes is not comparable to that for 4 processes. Hence, to obtain a valid comparison, we first identify the average convergence time for each of the programs. Subsequently, we use a fraction of this worst case requirement as the value of desired average convergence time. Specifically, we use three values: λ_1, λ_2 and λ_3, where λ_1 is 70% of the original average convergence time, λ_2 is 80% of the original convergence time and λ_3 is 90% of the original convergence time. We perform our experiments for $k \in \{3, 4, 5, 6, 7\}$, where k is the number of processes in the input program.

Our results are as shown in Tables 1 and 2. Specifically, Table 1 presents transition preservation percentage of original program for the revised program generated by our approaches. Table 2 presents revision time (in seconds) for generating the revised program that fits the λ requirements. In particular, we run each experiment for at most one hour. We set the running time as N/A when the experiment couldn't return a result within one hour. From these results, we find that SCP identifies the desired program most quickly. For example for 7 processes when requiring λ_3, SCP could find the desired program within 0.26 seconds. However, it eliminated most of the transitions. It only maintained 00.03 percentage of the original transitions. By contrast, KBP took significantly longer time. However, it kept 81.26 percentage of transitions. Observed from these results, for token-ring program, we find that RIAD provides the best approach for tradeoff between the time required to obtain the desired program and the number of transitions preserved in that program.

6.2 Stabilizing Algorithm Based on Raymond's Tree based Mutual Exclusion Program

This program, $\mathcal{P}_{rt}$, consists of n processes, numbered $0..(n-1)$. These processes are arranged in a fixed binary tree. For each process p_i, it has one variable h_i with domain$\{i, NBR_i\}$, where NBR_i denotes the neighbor processes of p_i. When $h_i = i$, then process p_i has the token. Otherwise, the holder of p_i points to one of its neighbors. In particular, $\mathcal{P}_{rt}$ provides three types of convergence actions as follows.

$$\begin{array}{llcl}
Action_0: & h_i \neq NBR_i \cup \{i\} & \longrightarrow & h_i = PR_i|i|NBR_i; \\
Action_1: & h_i \neq PR_i \wedge h_{PR_i} \neq i & \longrightarrow & h_i = PR_i; \\
Action_2: & h_i = PR_i \wedge h_{PR_i} = i & \longrightarrow & h_{PR_i} = PR_i;
\end{array}$$

In the above actions, PR_i denotes the parent process of p_i in the static tree and NBR_i denotes the neighbor processes of p_i. Specifically, the first action $Action_0$ ensures that the holder of a process points to its neighbors or itself. This action is executed by all processes. The second and third actions are exeted by all processes except the root process. Of these, the second action ensures that the holder of p_i is either PR_i or holder of PR_i is same as i. And, the third action ensures that the holder relation between p_i and PR_i is acyclic.

Table 1 Transition Coverage Percentage of Different Apporaches for K-state Token Ring Program.

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	3	46.67%	**80.00%**	51.00%	60.00%	48.57%	**80.00%**
0.7	4	16.25%	**80.00%**	60.16%	70.00%	62.50%	**80.00%**
0.7	5	02.96%	63.62%	53.36%	55.97%	**74.70%**	63.62%
0.7	6	00.37%	53.96%	33.48%	64.58%	**79.57%**	53.96%
0.7	7	00.03%	46.91%	26.50%	53.85%	**82.60%**	46.91%
0.8	3	46.67%	**93.33%**	51.11%	60.00%	48.57%	80.00%
0.8	4	16.25%	**88.75%**	68.60%	**92.50%**	**92.50%**	80.00%
0.8	5	02.96%	**81.84%**	65.29%	77.81%	74.70%	71.53%
0.8	6	00.37%	72.76%	60.01%	64.58%	**79.57%**	61.00%
0.8	7	00.03%	64.85%	55.70%	70.11%	**82.60%**	59.27%
0.9	3	46.67%	**93.33%**	82.22%	**94.29%**	**94.29%**	80.00%
0.9	4	16.25%	**95.63%**	85.47%	**92.50%**	**92.50%**	80.00%
0.9	5	02.96%	91.11%	84.69%	**99.57%**	**99.57%**	63.62%
0.9	6	00.37%	88.16%	83.07%	**99.44%**	**99.44%**	60.99%
0.9	7	00.03%	**83.93%**	**81.26%**	**84.67%**	**82.60%**	53.76%

Table 2 Revision Time (in seconds) of Different Approaches for K-state Token Ring Program.

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.7	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.7	5	<0.01	0.06	0.20	0.07	0.06	0.07
0.7	6	0.02	3.34	9.05	0.50	0.43	3.30
0.7	7	0.26	N/A	1,134.79	4.30	2.60	N/A
0.8	3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.8	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.8	5	<0.01	0.04	0.16	0.05	0.06	0.06
0.8	6	0.024	1.93	6.34	0.50	0.43	2.80
0.8	7	0.26	N/A	905.66	3.83	2.58	N/A
0.9	3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.9	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.9	5	<0.01	0.02	0.09	0.03	0.04	0.07
0.9	6	0.02	0.95	4.13	0.29	0.34	2.71
0.9	7	0.25	206.50	580.12	2.80	2.77	N/A

Table 3 Transition Coverage of Different Apporaches(Raymond Tree Based Mutual Exclusion Program).

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	4	3.13%	**86.03%**	**82.91%**	58.46%	58.46%	77.67 %
0.7	5	0.38%	**81.25%**	**83.73%**	43.02%	58.46%	**81.25 %**
0.7	6	00.03 %	**85.60%**	84.43%	38.97%	52.06 %	**81.96%**
0.7	7	<0.01%	**86.22%**	**85.60%**	33.91%	49.66%	**84.40%**
0.7	8	<0.01%	**87.20 %**	**87.81 %**	34.14%	39.00%	**86.25%**
0.8	4	3.13%	**77.67%**	**75.83%**	49.63%	53.66%	**77.67 %**
0.8	5	0.38%	**85.66%**	**83.73%**	55.12%	60.98%	**81.25 %**
0.8	6	00.03%	**85.60%**	**87.73%**	40.44%	57.35 %	81.96%
0.8	7	<0.01%	**88.39%**	87.37%	36.04%	52.70 %	**84.18%**
0.8	8	<0.01%	**88.37%**	**89.43%**	35.00%	49.90%	**86.23%**
0.9	4	3.13%	86.03%	**91.64%**	76.47%	76.47%	77.67 %
0.9	5	0.38%	**91.31%**	**89.13%**	55.12%	63.41%	78.61 %
0.9	6	0.03%	**89.22%**	**91.89%**	62.21%	58.82%	81.96%
0.9	7	<0.01%	**91.42%**	**93.58%**	61.92%	57.77%	84.18%
0.9	8	<0.01%	**92.88%**	**91.87%**	63.85%	53.94%	86.23%

Table 4 Revision Time (in seconds) of Different Approaches (Raymond Tree Based Mutual Exclusion Program).

λ	# proc	SCP	ELP	KBP	RIA	RIAD	SSP
0.7	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.7	5	<0.01	0.01	0.03	0.03	0.02	0.01
0.7	6	0.02	0.25	1.11	1.08	0.71	0.30
0.7	7	0.15	11.87	53.52	49.80	30.53	12.67
0.7	8	0.01	17.47	1,741.98	3,249.34	2,180.22	24.73
0.8	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.8	5	<0.01	0.01	0.03	0.02	0.02	0.01
0.8	6	0.02	0.24	1.14	1.00	0.69	0.30
0.8	7	0.16	9.45	50.18	47.77	32.58	12.67
0.8	8	0.01	12.59	1,706.08	3,170.72	1,905.08	24.54
0.9	4	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
0.9	5	<0.01	<0.01	0.02	0.02	0.01	0.01
0.9	6	0.02	0.19	0.95	0.67	0.58	0.30
0.9	7	0.16	6 .35	32.63	42.37	28.80	12.67
0.9	8	0.01	5.39	1,678.03	1,845.25	1,845.94	24.54

Tables 3 and 4 present our experiment results for Raymond Tree based mutual exclusion program. In particular, Table 3 illustrates the transition coverage percentage of the original program for the newly generated program with respect to our six approaches. Table 4 shows the performance of these six approach in revision time (in seconds). Similar to previous study, we run each experiment for at most one hour. If the revision time exceeds one hour, we will identify it as N/A in the table.

We perform our experiments for $n \in \{4, 5, 6, 7, 8\}$, where n is the number of processes in the input program. From these results, same as in the experiment for token-ring program, SCP identifies the desired program most quickly. For example for 7 processes when λ is set to 0.9, SCP could find the desired program within 0.25 seconds. However, it eliminated most of the transitions. It maintained less than 0.01 percentage of the original transitions. By contrast, SSP took significantly longer time. However, it kept 86.23 percentage of transitions. Observed from these results, we find that KBP provide the best approach for tradeoff between the time required to obtain the desired program and the number of transitions preserved in that program.

7 Related Work

In this work, we focused on the problem of adding average recovery in the presence of an adversarial scheduler. The closest comparable work to this is [1] where authors have considered the problem of synthesizing a program with given average recovery time. In this work, the authors omit the notion of an adversarial scheduler. Instead, they assume that each non-deterministic choice is resolved through randomization. Hence, if the program synthesized using these approaches is used to reduce the recovery time then its average convergence time in the presence of an adversary can be higher. By contrast, in our work, we have focused on the problem of guaranteeing average recovery time in the presence of an adversary. In other words, the solution provided by our approaches will ensure that even if the adversary puts arbitrary probabilities on different non-deterministic choices, the average recovery constraint will be satisfied.

The work in [7, 12, 19, 6, 14, 18, 20, 22, 16, 13, 15] has focused on the topic of adding safety properties, liveness properties and fault-tolerance properties. The properties considered in this work are represented using the framework of safety and liveness by Alpern and Schneider [2]. As discussed in Section 1, each program computation can be evaluated independently to determine whether it satisfies or violates the specification. By contrast, the average response time considered in this paper cannot be represented using the framework in [2]. It requires a more generalized framework of hyperproperties [9]. In this framework, satisfaction of a requirement is determined by *all* computations included by the program. In particular, the average convergence time is an instance of a hyperliveness property. While the work in this paper enables one to repair a given program to add one hyperliveness property, one future work in this area is to generalize to other hypersafety and hyperliveness properties.

8 Conclusion

We focused on the problem of revising a given program to add average recovery time in the presence of an adversarial scheduler who could force the program to choose the least desirable path during recovery. Adding average recovery time requires removal of some behaviors/transitions that cause the recovery to increase beyond acceptable limit. We showed that ensuring that only a minimum number of transitions are removed is NP-hard. Hence, we proposed six different heuristics.

We find that, as expected, the first heuristic, SCP, constructs the desired program in the least amount of time. However, it ends up removing a large number of transitions unnecessarily. For example, in case of the token ring program with 7 processes, it found the desired program in 0.2 seconds. However, it preserved only 00.03 percent of transitions. By contrast, RIAD preserved 82.60% of transitions but took around 2 seconds to obtain the desired program.

We presented the analysis of our six approaches in two case studies. Based on these case studies, we find that RIAD and KBP provide the best approaches for tradeoff between the time required to obtain the desired program and the number of transitions preserved in that program. We plan to conduct more case studies in the future so that we could identify the effect of specific program structure on program revision for the problem of average recovery time.

In our work, we focused on the problem of average recovery time in the presence of an adversarial scheduler. There are two aspects to the recovery in the presence of faults: (1) state to which the program is perturbed to when faults stop occurring, and (2) the non-deterministic choices made by the scheduler during recovery. Regarding the first aspect, we considered the case where the state to which the program is perturbed to is chosen with equal probability. However, it is straightforward to extend it to the case where each state is associated with a different probability distribution. This will only change the way average convergence time is computed. However, all our aprpaoches could still be used. Regarding the second aspect, we assumed that the scheduler can arbitrarily choose the execution order. Our work could also be extended to other choices of scheduler.

This work also demonstrates the feasibility of adding some hyperproperties [9]. Specifically, the requirement of average convergence time cannot be expressed in terms of the framework of safety and liveness by [2]. This is due to the fact that checking whether a given program computation is acceptable or not depends upon other computations involved in the program. A possible future work in this area is to pursue such repair for other hyperproperties.

References

1 Saba Aflaki, Fathiyeh Faghih, and Borzoo Bonakdarpour. Synthesizing self-stabilizing protocols under average recovery time constraints. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 – July 2, 2015*, pages 579–588, 2015.

2 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

3 A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

4 Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011. doi:10.1017/S0960129511000193.

5 Borzoo Bonakdarpour. *Automated Revision of Distributed and Real-Time Programs*. PhD thesis, Michigan State University, 2009.

6 Borzoo Bonakdarpour, Ali Ebnenasir, and Sandeep S. Kulkarni. Complexity results in revising unity programs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(1):5:1–5:28, Feb. 2009.

7 Borzoo Bonakdarpour, Sandeep S. Kulkarni, and Fuad Abujarad. Symbolic synthesis of masking fault-tolerant distributed programs. *Distributed Computing*, 25(1):83–108, 2012. doi:10.1007/s00446-011-0139-3.

8 Buddy – a binary decision diagram package. `http://buddy.sourceforge.net/manual/main.html`.

9 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. `doi:10.3233/JCS-2009-0393`.

10 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. `doi:10.1145/361179.361202`.

11 Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

12 Ali Ebnenasir and Aly Farahat. Swarm synthesis of convergence for symmetric protocols. In Cristian Constantinescu and Miguel P. Correia, editors, *EDCC*, pages 13–24. IEEE, 2012. `doi:10.1109/EDCC.2012.22`.

13 Ali Ebnenasir and Aly Farahat. Swarm synthesis of convergence for symmetric protocols. In *Proceedings of the Ninth European Dependable Computing Conference*, pages 13–24, 2012.

14 Ali Ebnenasir and Sandeep S. Kulkarni. Feasibility of stepwise design of multitolerant programs. *ACM Trans. Softw. Eng. Methodol.*, 21(1):1, 2011. `doi:10.1145/2063239.2063240`.

15 Aly Farahat and Ali Ebnenasir. A lightweight method for automated design of convergence in network protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):38:1–38:36, December 2012.

16 Aly Farahat and Ali Ebnenasir. Local reasoning for global convergence of parameterized rings. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 496–505, 2012.

17 Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

18 Alex Klinkhamer and Ali Ebnenasir. A software tool for swarm synthesis of self-stabilization. `http://www.cs.mtu.edu/~apklinkh/protocon/index.html`.

19 Alex Klinkhamer and Ali Ebnenasir. On the complexity of adding convergence. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 8161 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2013. `doi:10.1007/978-3-642-40213-5_2`.

20 Alex Klinkhamer and Ali Ebnenasir. Synthesizing self-stabilization through superposition and backtracking. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 252–267. Springer, 2014.

21 K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

22 Amer Tahat and Ali Ebnenasir. A hybrid method for the verification and synthesis of parameterized self-stabilizing protocols. In *24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, 2014.

Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures*

Christian Scheideler[1], Alexander Setzer[2], and Thim Strothmann[3]

1 **Paderborn University, Paderborn, Germany**
2 **Paderborn University, Paderborn, Germany**
3 **Paderborn University, Paderborn, Germany**

Abstract

Distributed applications are commonly based on overlay networks interconnecting their sites so that they can exchange information. For these overlay networks to preserve their functionality, they should be able to recover from various problems like membership changes or faults. Various self-stabilizing overlay networks have already been proposed in recent years, which have the advantage of being able to recover from any illegal state, but none of these networks can give any guarantees on its functionality while the recovery process is going on. We initiate research on overlay networks that are not only self-stabilizing but that also ensure that searchability is maintained while the recovery process is going on, as long as there are no corrupted messages in the system. More precisely, once a search message from node u to another node v is successfully delivered, all future search messages from u to v succeed as well. We call this property *monotonic searchability*. We show that in general it is impossible to provide monotonic searchability if corrupted messages are present in the system, which justifies the restriction to system states without corrupted messages. Furthermore, we provide a self-stabilizing protocol for the line for which we can also show monotonic searchability. It turns out that even for the line it is non-trivial to achieve this property. Additionally, we extend our protocol to deal with node departures in terms of the Finite Departure Problem of Foreback et al. (SSS 2014). This makes our protocol even capable of handling node dynamics.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Topological Self-Stabilization, Monotonic Searchability, Node Departures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.24

1 Introduction

The Internet has opened up tremendous opportunities for people to interact and exchange information. Particularly popular ways to interact are peer-to-peer systems and social networks. For these systems to stay popular, it is very important that they are highly available. However, once these systems become large enough, changes and faults are not an exception but the rule. Therefore, mechanisms are needed that ensure that whenever there are problems, they are quickly repaired, and all parts of the system that are still functional should not be affected by the repair process. Protocols that are able to recover from arbitrary states are also known as *self-stabilizing* protocols.

Since the seminal paper of Dijkstra in 1974 [4], self-stabilizing protocols have been investigated for many classical problems including leader election, consensus, matching,

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 24; pp. 24:1–24:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

clock synchronization and token distribution problems. Recently, also various protocols for self-stabilizing overlay networks have been proposed (e.g., [14, 9, 6, 10, 5, 1, 11, 12, 2]). However, for all of these protocols it is only known that they *eventually* converge to the desired solution, but the convergence process is not necessarily *monotonic*. In other words, it is not ensured for two points in time t, t' with $t < t'$ that the functionality of the topology at time t' is better than the functionality at time t.

In this paper, we focus on protocols for self-stabilizing overlay networks that guarantee the *monotonic* preservation of a characteristic that we call *searchability*, i.e., once a search message from node u to another node v is successfully delivered, all future search messages from u to v succeed as well. Searchability is a useful and natural characteristic for an overlay network since searching for other participants is one of the most common tasks in real-world networks. Moreover, a protocol that preserves monotonic searchability has the huge advantage that in every state, even if the self-stabilization process has not converged yet, the already built topology can already be used for search requests.

As a starting point for rigorous research on monotonic searchability, we will focus on building a self-stabilizing protocol that preserves monotonic searchability for the line graph. Although the topology itself is fairly simple, to preserve searchability during the self-stabilization process turns out to be quite challenging. Additionally, we study monotonic searchability for the line graph if the node set is dynamic, i.e., nodes are allowed to leave the network.

1.1 Model

We consider a distributed system consisting of a fixed set of nodes in which each node has a unique reference and a unique immutable numerical identifier (or short id). The system is controlled by a protocol that specifies the variables and actions that are available in each node. In addition to the protocol-based variables there is a system-based variable for each node called *channel* whose values are sets of messages. We denote the channel of node u as $u.Ch$ and $u.Ch$ contains all incoming messages to u. Its message capacity is unbounded and messages never get lost. A node can add a message to $u.Ch$ if it has a reference to u. Besides these channels there are no further communication means, so only point-to-point communication is possible.

There are two types of actions. The first type of *action* has the form of a standard procedure $\langle label\rangle(\langle parameters\rangle)$: $\langle command\rangle$, where *label* is the unique name of that action, *parameters* specifies the parameter list of the action, and *command* specifies the statements to be executed when calling that action. Such actions can be called remotely. In fact, we assume that every message must be of the form $\langle label\rangle(\langle parameters\rangle)$ where *label* specifies the action to be called in the receiving node and *parameters* contains the parameters to be passed to that action call. All other messages will be ignored by the nodes. Apart from being triggered by messages, these actions may also be called locally by the nodes, which causes their immediate execution. The second type of action has the form $\langle label\rangle : \langle guard\rangle \longrightarrow \langle command\rangle$, where *label* and *command* are defined as above and *guard* is a predicate over local variables. We call an action whose guard is simply **true** a *timeout* action.

The *system state* is an assignment of a value to every variable of each node and messages to each channel. An action in some node p is *enabled* in some system state if its guard evaluates to **true**, or if there is a message in $p.Ch$ requesting to call it. In the latter case the corresponding message is processed (in which case it is removed from $p.Ch$). An action is *disabled* otherwise. Receiving and processing a message is considered as an atomic step.

A *computation* is an infinite fair sequence of system states such that for each state s_i, the next state s_{i+1} is obtained by executing an action that is enabled in s_i. This disallows the overlap of action execution. That is, action execution is *atomic*. We assume *weakly fair action execution* and *fair message receipt*. Weakly fair action execution means that if an action is enabled in all but finitely many states of the computation, then this action is executed infinitely often. Note that the timeout action of a node is executed infinitely often. Fair message receipt means that if the computation contains a state where there is a message in a channel of a node that enables an action in that node, then that action is eventually executed with the parameters of that message, i.e., the message is eventually processed. Besides these fairness assumptions, we place no bounds on message propagation delay or relative nodes execution speeds, i.e., we allow fully asynchronous computations and non-FIFO message delivery. A *computation suffix* is a sequence of computation states past a particular state of this computation. In other words, the suffix of the computation is obtained by removing the initial state and finitely many subsequent states. Note that a computation suffix is also a computation.

We consider protocols that do not manipulate the internals of node references. Specifically, a protocol is *compare-store-send* if the only operations that it executes on node references is comparing them, storing them in local memory and sending them in a message. That is, operations on references such as addition, radix computation, hashing, etc. are not used. In a compare-store-send protocol, if a node does not store a reference in its local memory, the node may learn this reference only by receiving it in a message. A compare-store-send protocol cannot introduce new references to the system. It can only operate on the references that are already there.

The overlay network of a set of nodes is determined by their knowledge of each other. We say that there is a (directed) *edge* from a to b, denoted by (a, b), if node a stores a reference of b in its local memory or has a message in $a.Ch$ carrying the reference of b. In the former case, the edge is called *explicit* (drawn solid in figures), and in the latter case, the edge is called *implicit* (drawn dashed). With NG we denote the directed *network (multi-)graph* given by the explicit and implicit edges. ENG is the subgraph of NG induced by only the explicit edges. A *weakly connected component* of a directed graph G is a subgraph of G of maximum size so that for any two nodes u and v in that subgraph there is a (not necessarily directed) path from u to v. Two nodes that are not in the same weakly connected component are *disconnected*. We say a node a is to the *left* (*right*, respectively) of a node b if $id(a) < id(b)$ ($id(a) > id(b)$). If there is an edge (a, b) between the two, then a is a *left neighbor* (*right neighbor*). For three nodes a, b, c with $id(a) < id(b), id(a) < id(c)$ (or $id(a) > id(b), id(a) > id(c)$, respectively), we say a node b is *closer* to a than c, if $|id(a) - id(b)| < |id(a) - id(c)|$. If it is clear from the context we sometimes refer to the identifier of a node by dropping the id notation to , e.g., we write $a < b$ instead of $id(a) < id(b)$.

In this paper we are particularly concerned with search requests, i.e., Search$(v, destID)$ messages that are routed along ENG according to a given routing protocol, where v is the sender of the message and $destID$ is the identifier of a node we are looking for. Note that $destID$ does not necessarily belong to an existing node w, since we also want to model search requests to not existing nodes. If a Search$(v, destID)$ message reaches a node w with $id(w) = destID$, the search request *succeeds*; if the message reaches some node u with $id(u) \neq destID$ and cannot be forwarded anymore according to the given routing protocol, the search request *fails*. We assume that nodes themselves initiate Search() requests at will. Therefore, the Search$(destID)$ action is never explicitly called.

We need some additional notation for our results of Section 4, in which we extend the protocol to handle nodes that want to leave the system. A node u has a variable *mode* $\in$ {leaving, staying} that is read-only. If this variable is set to **leaving**, the node is *leaving*; the node is *staying* if the variable is set to **staying**. Note that staying nodes can dynamically decide at any arbitrary state if they want to leave the system by executing a corresponding *leave action*. However, a leaving node cannot switch back to staying. The ultimate goal of a leaving node is to depart from the system. There is one special command that is important for the study of leaving nodes: **exit**. If a node executes **exit** it enters a designated *exit state* and all remaining edges to or from that node are deleted. We call such a node *gone*. A node that is not gone is called *present*. For a gone node all actions are disabled, in particular it will not execute the timeout action regularly.

1.2 Problem Statement

A protocol is *self-stabilizing* if it satisfies the following two properties.

Convergence: starting from an arbitrary system state, the protocol is guaranteed to arrive at a legitimate state.

Closure: starting from a legitimate state the protocol remains in legitimate states thereafter.

A self-stabilizing protocol is thus able to recover from transient faults regardless of their nature. Moreover, a self-stabilizing protocol does not have to be initialized as it eventually starts to behave correctly regardless of its initial state. In *topological self-stabilization* we allow self-stabilizing protocols to perform changes to the overlay network, resp. NG. A legitimate state may then include a particular graph topology or a family of graph topologies.

In this paper we want to build a self-stabilizing protocol for the *linearization problem*, i.e., the nodes are sorted by identifiers and each node stores only two references: its closest successor and its closest predecessor. From a global point of view, the nodes build a *line graph* topology. Of course, searching is easy once a legitimate state has been reached. However, searching reliably during the stabilization phase is much more involved. We say a (self-stabilizing) protocol satisfies *monotonic searchability* according to some routing protocol R if it holds for any pair of nodes v, w that once a $\textsc{Search}(v, id(w))$ request (that is routed according to R) initiated at time t succeeds, any $\textsc{Search}(v, id(w))$ request initiated at a time $t' > t$ will succeed. We do not mention R if it is clear from the context. A protocol is said to satisfy *non-trivial* monotonic searchability if it satisfies monotonic searchability and in every computation of the protocol there is a suffix such that for each pair of nodes v, w for which there is a path from v to w in the target topology $\textsc{Search}(v, id(w))$ requests will succeed.

Furthermore, we give a self-stabilizing protocol that satisfies non-trivial monotonic searchability, solves the linearization problem and solves the *Finite Departure Problem* of [7]. The following problem statement is adapted from [13]:

Finite Departure Problem ($\mathcal{FDP}$): In case the **exit** command is available, eventually reach a system state in which (i) every staying node is awake, (ii) every leaving node is gone and (iii) for each weakly connected component of the initial network graph, the staying nodes in that component still form a weakly connected component.

Consequently, a leaving node u should *safely* execute **exit**, i.e., the removal of u and its incident edges from NG does not disconnect any present nodes and does not violate searchability.

1.3 Related work

The idea of self-stabilization in distributed computing was introduced in a classical paper by E. W. Dijkstra in 1974 [4], in which he looked at the problem of self-stabilization in a token ring. In order to recover certain network topologies from any weakly connected state, researchers started with simple line and ring networks (e.g. [17, 15, 8]. Over the years more and more network topologies were considered, ranging from skip lists and skip graphs [14, 9], to expanders [6], Delaunay graphs [10], hypertrees and double-headed radix trees [5, 1], small-world graphs [11] and a Chord variant [12]. Also a universal algorithm for topological self-stabilization is known [2].

Close to our work is the notion of *monotonic convergence* by Yamauchi and Tixeuil [18]. A self-stabilizing protocol is monotonically converging if every change done by a node p makes the system approach a legitimate state and if every node changes its output only once. The authors investigate monotonically converging protocols for different classic distributed problems (e.g., leader election and vertex coloring) and focus on the amount of non-local information that is needed for them.

Our study of the *Finite Departure Problem* is heavily inspired by [7], in which the authors propose the aforementioned problem to study graceful departures of nodes in a self-stabilizing setting, i.e., nodes that want to leave a distributed system should decide when they can leave without affecting weak connectivity of the topology. They conclude that in general it is not possible to solve the $\mathcal{FDP}$. However, with the use of distributed oracles (which are specialized failure detectors [3]) the authors propose a protocol that solves the problem and arranges the nodes in a line. Additionally, they can show that oracles are not needed if the problem is transformed into a non-decision variant. In [13] the idea is generalized to a protocol framework that solves the $\mathcal{FDP}$ without being reliant on a certain topology and is thereby combinable with most existing overlay protocols.

1.4 Our contribution

To the best our knowledge, this paper presents the first attempt to have stricter requirements towards the self-stabilization process in topological self-stabilization. We define and study *monotonic searchability*, which captures a typical use case for overlay networks, i.e., searching other nodes. More formally, we want to guarantee for a self-stabilizing topology that once a search message from node u to another node v is successfully delivered, all future search messages from u to v succeed as well. We focus on studying non-trivial monotonic searchability for the list topology. First, we show that in general it is impossible to provide non-trivial monotonic searchability from any initial system state, due to the presence of certain initial messages. This justifies to study searchability only for so-called *admissible system states* in which these messages are not present anymore, as long as the protocol gurantees convergence to these states. We give a self-stabilizing list protocol and an appropriate search protocol that achieve the desired goal and prove their correctness. Moreover, we broaden the elaborateness of the problem statement, by allowing nodes to leave the line topology, i.e., solving the Finite Departure Problem in addition to the aforementioned problems. Also for this combination of problems we present suitable protocols and prove their correctness.

2 Preliminaries

Since gone nodes will never execute any action, we only consider initial states in which all nodes are present. We also restrict the initial state to contain only a finite number of messages that can trigger actions specified by our protocol, since other messages are ignored by the nodes. Finally, we do not allow the presence of references that do not belong to a node in the system. From now on, an initial system state satisfies all of these constraints.

The following propositions are restatements of results in [14] and imply further necessary conditions on initial system states.

1. If a compare-store-send program solves the linearization problem, each computation starts in a weakly connected initial state.
2. If a compare-store-send program solves the linearization problem, each computation starts in a state in which all references belong to present nodes.

A *message invariant* is a predicate of the following form: If there is a message m in the incoming channel of a node, then a predicate P' must hold. A protocol may specify one or more message invariants. An arbitrary message m in a system is called *corrupted* if the existence of m violates one of the message invariants. A state s is called *admissible* if there are no corrupted messages in s. We say a protocol *admissible-message satisfies* a property if the following two conditions hold: (i) in computations in which every state is admissible, it satisfies the property, and (ii) starting from any initial state, there is a computation suffix in which every state is admissible. A protocol *unconditionally satisfies* a property if it satisfies this property starting from any state.

With this notion in mind, we can show that admissible-message satisfaction is necessary for non-trivial monotonic searchability for any routing algorithm R.

▶ **Lemma 1.** *If a compare-store-send self-stabilizing protocol satisfies non-trivial monotonic searchability then this protocol must be admissible-message satisfying.*

The structure of the proof is as follows: we consider an arbitrary unconditionally satisfying protocol and show that it does not satisfy monotonic searchability by creating a bad instance for this protocol. In particular, we exploit that our model does not ensure FIFO delivery of messages. The proof can be found in the full version of this paper [16].

Consequently, to prove non-trivial monotonic searchability for a protocol (according to a given routing protocol R) it is sufficient to show that: (i) the protocol has a computation suffix in which every state is admissible and (ii) the protocol guarantees non-trivial monotonic searchability according to R in admissible states.

For the $\mathcal{FDP}$, it was shown in [7], there is no distributed protocol within our model that can decide when it is safe for a node u to leave the system and thereby solve the $\mathcal{FDP}$. The authors circumvent this impossibility result with the help of oracles. In general, an *oracle* is a predicate that depends on the current system state and the node calling it. In the context of the $\mathcal{FDP}$, an oracle is supposed to advise a leaving node when it is safe to execute **exit**. We use the oracle $\mathcal{NIDEC}$ as introduced in [7] in order to solve the $\mathcal{FDP}$. $\mathcal{NIDEC}$ evaluates to **true** for a node u calling it, if no node $v \neq u$ has a reference to u in its local memory or in a message in $v.Ch$ and if $u.Ch$ is empty. For an in depth discussion of oracles for the $\mathcal{FDP}$, we refer the reader to [7, 13].

3 The Build-List+ and the Search+ protocols

In this section, we present the Build-List+ protocol and the Search+ protocol. Build-List+ solves the linearization problem and is admissible-message satisfying non-trivial monotonic searchability according to Search+. Note that any protocol satisfying non-trivial monotonic searchability must be admissible-message satisfying as shown in Section 2. This section is organized as follows: First, we describe Build-List+ and Search+ in detail (Subsection 3.1). Then, we prove that the Build-List+ protocol solves the linearization problem (Subsection 3.2). Last, we prove that the Build-List+ protocol satisfies non-trivial monotonic searchability according to Search+ (Subsection 3.3). From now on we drop the "according to Search+" clause, since we only consider searchability for Search+.

3.1 Description of Build-List+ and Search+

The Build-List+ Protocol builds upon the protocol introduced in [15] that solves the linearization problem. For this protocol, every node only keeps a single left and right neighbor. If a node u receives a reference of a node v with $u < v$ ($u > v$, respectively), u either saves v as its new right (left) neighbor if v is closer to u than the current right (left) neighbor w and delegates the reference of w to v or (in case v is not closer), v is not saved and delegated to w. Here, *delegation* means that the reference of a node is sent in a message to another node and not kept in the local memory. A natural (local) search protocol for this topology is to always forward search requests to the neighbor closest to the desired target node, or to abort the search request in case no such neighbor exists. Note that these easy and elegant protocols cannot guarantee monotonic searchability due to three simple facts: (i) due to delegation, it is possible that an explicit edge (u, v) is replaced by an explicit edge (u, w) and an implicit edge (w, v), (ii) consequently, u, v are not in the same weakly connected component in ENG (even though they were before delegation) and (iii) searchability is defined for ENG.

The Build-List+ protocol introduces the following changes in order to satisfy monotonic searchability: Instead of having a single left and right neighbor, a node u has sets of neighbors *Left* and *Right* (that it sorts implicitly according to id). In the following, whenever we use the notation $Left(u)/Right(u)$, we refer to these sets of a node u. The main principle that we use is that every node w does not delegate any edge to a node v stored in $Left(w)$ or $Right(w)$ directly. Instead it first introduces (using Introduce(v, w)) this node to another node u, waits for an acknowledgement that the edge has been added to $Left(u)$ or $Right(u)$ (which is basically the Linearize(v) message) and then delegates the edge to a node closer to v (using TempDelegate(v)). More specifically, whenever a node u has multiple neighbors to one side, it does not delegate edges to the closest neighbor directly, but does the following. W.l.o.g. assume that it has multiple neighbors $w_1, \dots, w_\ell$ to the right with $id(w_i) < id(w_{i+1})$. In the Timeout action u introduces w_i to w_{i-1}, with an Introduce(w_i, u) message. Thereby, w_{i-1} knows that it got the reference from u, saves the reference to w_i directly, sends a Linearize(w_i) message back to u and a TempDelegate(u) to itself (the latter is only to preserve connectivity). Node u can now react to that Linearize(w_i) message, by deleting w_i from its memory and sending the reference to the closest node to the left of w_i in *Right* (which is not necessarily w_{i-1} anymore). Thereby, u preserves a path of explicit edges between u and w_i. Additionally, u sends its own reference to the closest neighbors with a Introduce$(u, \perp)$ message who turn this into a TempDelegate(u) message. In general, the TempDelegate(u) action is used to delegate an implicit edge to a node u into one direction (i.e., to the left or to the right) as long as there is a node between the current node and u in *Left* or *Right*. Note that implicit edges are not used for search, thus we do not have to apply the principle of introducing first and delegating afterwards for this kind of edges. However, we have to delegate in order to preserve connectivity and to stabilize to the line eventually. Note that, even though a node has temporarily more references than necessary for the final line topology our protocol still eventually stabilizes to the line, as we will show later. The pseudocode for all Build-List+ actions is given in Listing 1. Note that a node refers to itself with the expression $self$. Additionally, keep in mind that the timeout action is the only action that is not triggered as a result of another action. Instead, is triggered regularly.

Listing 1 Build-List+ protocol

```
TIMEOUT
 for all $destID \in Waiting$
   send $forwardProbe(self, destID, \{self\}, self.seq)$ to $self$
 //Let $Left = \{v_1, v_2, \ldots, v_k\}$ with $id(v_1) < id(v_2) < \cdots < id(v_k)$
 for all $v_i \in Left$ with $1 \leq i < k$
   send INTRODUCE($v_i, self$) to $v_{i+1}$
 //Let $Right = \{w_1, w_2, \ldots, w_l\}$ with $id(w_1) < id(w_2) < \cdots < id(w_l)$
 for all $w_i \in Right$ with $1 < i \leq l$
   send INTRODUCE($w_i, self$) to $w_{i-1}$
 send INTRODUCE($self, \perp$) to $v_1$
 send INTRODUCE($self, \perp$) to $w_1$

INTRODUCE($v, w$)
 if ($id(v) < id(self)$)
   if{$w \neq \perp$}
     $Left \leftarrow Left \cup \{v\}$
     send LINEARIZE($v$) to $w$
     send TEMPDELEGATE($w$) to $self$
   else //$w = \perp$
     send TEMPDELEGATE($v$) to $self$
 else if ($id(v) > id(self)$)
   //Analogous to the previous case.

LINEARIZE($v$)
 send TEMPDELEGATE($v$) to $self$
 if ($id(v) < id(self)$)
   if ($Left \neq \emptyset$)
     $x \leftarrow argmax\{id(x')|x' \in Left\}$
     if ($v \neq x$)
       $w \leftarrow argmin\{id(w')|w' \in Left\ \ und\ \ id(w') > id(v)\}$
       $Left \leftarrow Left \setminus \{v\}$
       send TEMPDELEGATE($v$) to $w$
 else if ($id(v) > id(self)$)
   //Analogous to the previous case.

TEMPDELEGATE($u$)
 if ($id(u) < id(self)$)
   if ($Left = \emptyset$)
     $Left \leftarrow Left \cup \{u\}$
   else //$Left \neq \emptyset$
     $x \leftarrow argmax\{id(x')|x' \in Left\}$
     if ($id(x) < id(u)$)
       $Left \leftarrow Left \cup \{u\}$
     else if ($id(x) > id(u)$)
       send TEMPDELEGATE($u$) to $x$
 else if{$id(u) > id(self)$}
   //Analogous to the previous case.
```

The SEARCH+ protocol works as follows: Whenever the INITIATENEWSEARCH($destID$) action is called at a node u, u creates a new SEARCH($u, destID$) message and starts to periodically initiate FORWARDPROBE($u, destID, \{u\}, self.seq$) messages that it sends to itself. In the following, assume $id(u) < destID$ (the other case is analogous). Each FORWARDPROBE() message has a set of nodes, called $Next$ attached to it, which contains the nodes the message will visit in its future. It also has a counter seq attached to it whose meaning we will explain later. Whenever a FORWARDPROBE($u, destID, Next, seq$) message is at a node w, w removes itself from $Next$ and adds all its right neighbors x with $id(x) \leq destID$ to $Next$. Then it forwards the FORWARDPROBE($u, destID, Next, seq$) message to the node with minimal id in $Next$. If a FORWARDPROBE($u, destID, Next, seq$) message arrives at a node v with $id(v) = destID$, it directly responds with a PROBESUCCESS($destID, seq, v$) message to u. However, if $Next$ is empty at a node w with $id(w) \neq destID$ after w has added the aforementioned right neighbors, the FORWARDPROBE() message is answered with a PROBEFAIL($destID, seq$) message. In any case, as soon as u receives the response, it acts accordingly: If the answer to a FORWARDPROBE($u, destID, Next, seq$) message is a PROBEFAIL($destID, seq$) message, it drops the corresponding SEARCH($u, destID$) message completely. If the answer is PROBESUCCESS($destID, v$), SEARCH($u, destID$) messages waiting at u are directly sent to v.

Note that if additional SEARCH($u, destID$) messages are created at u while u is still waiting for an answer to an earlier initiated FORWARDPROBE($u, destID$), these requests simply wait together with the previous request (realized by simple $WaitingFor[destID]$ field) and are aborted or sent as soon as the PROBEFAIL($destID$) or PROBESUCCESS($destID, v$) response arrives at u, (i.e., search requests to the same destination are sent out in batches if possible). Furthermore, note that nodes do not memorize whether they have already sent FORWARDPROBE() messages to a certain destination. Due to corrupt initial states, this knowledge could be wrong and nodes relying on this knowledge would wait forever. Therefore, nodes periodically send FORWARDPROBE() messages, instead of only once. Note that because we make no assumptions on the message delivery speed and channels are not FIFO, it is possible that PROBEFAIL() messages arrive at a node u that are answers to FORWARDPROBE() messages initiated long ago. However, in the meantime, there might have been successful responses. To deal with this, each node u stores a sequence number counter seq. Whenever INITIATENEWSEARCH($destID$) is executed by u and there is no SEARCH($u, destID$) that waits for an answer to a FORWARDPROBE($u, destID, Next, seq$) message, u increments $u.seq$, stores the new $u.seq$ value in an entry for v and always attaches the current sequence number ($u.seq$) to each FORWARDPROBE() message u sends. Responses to probes (success and failure) sent by u also contain this sequence number. Whenever a response is sent back to u, u checks whether the sequence number in this message is at least the sequence number stored for $destID$. If not, it simply drops the message, since in that case, the answer belongs to a FORWARDPROBE() message sent for an earlier batch of SEARCH($u, destID$) messages that have already been processed. The complete pseudocode for SEARCH+ is given in Listing 2.

In order to not unnecessarily blow up the pseudocode, we intentionally left out a sanity check for each node, i.e., before executing each action, each node u makes sure that $Left$ only contains nodes v with $v < u$ and that $Right$ only contains nodes v with $u < v$. If this is not the case for some node v, u rearranges the reference to v accordingly. This way, in every computation, the following lemma holds:

▶ **Lemma 2.** *For every node v it holds: For all $x \in Left$, $id(x) < id(v)$, and for all $y \in Right$, $id(v) < id(y)$.*

Listing 2 SEARCH+ protocol

```
INITIATENEWSEARCH($destID$)
 create new message $m = $ SEARCH($self, destID$)
 if ( $WaitingFor[destID] = \emptyset$ )
   $WaitingFor[destID] \leftarrow \{\}$
   $self.seq \leftarrow self.seq + 1$
   $seq[destID] \leftarrow self.seq$
 //Store the messages to $WaitingFor$
 $WaitingFor[destID] \leftarrow WaitingFor[destID] \cup \{m\}$

FORWARDPROBE($source, destID, Next, seq$)
 if ( $destID = id(self)$ )
   if ( $Next \neq \emptyset$ )
     for all $u \in Next$
       send TEMPDELEGATE($u$) to $self$
   send PROBESUCCESS($destID, seq, self$) to $source$
   send TEMPDELEGATE($source$) to $self$
 else //$destID \neq id(self)$
   if ( $destID > id(self)$ )
     $Next \leftarrow Next \setminus \{self\} \cup \{w \in Right | id(w) \leq destID\}$
     if ( $Next = \emptyset$ )
       send PROBEFAIL($destID, seq$) to $source$
       send TEMPDELEGATE($source$) to $self$
     else //$Next \neq \emptyset$
       $u \leftarrow argmin\{id(u) | u \in Next\}$
       if ( $id(u) < id(self)$ )
         send TEMPDELEGATE($u$) to $self$
       else if ( $id(u) < id(argmin\{id(v) | v \in Right\})$ )
         $Right \leftarrow Right \cup \{u\}$
       send FORWARDPROBE($source, destID, Next, seq$) to $u$
   else if ( $destID < id(self)$ )
     //Analogous to the previous case.

PROBESUCCESS($destID, seq, dest$)
 if ( $seq \geq seq[destID]$ )
   /* The message belongs to currently
    * stored search requests to $dest$. */
   send all $m \in WaitingFor[destID]$ to $dest$
   $WaitingFor[destID] \leftarrow \emptyset$
 send TEMPDELEGATE($dest$) to $self$

PROBEFAIL($destID, seq$)
 if ( $seq \geq seq[destID]$ )
   /* The message belongs to currently
    * stored search requests to $dest$. */
   $WaitingFor[destID] \leftarrow \emptyset$
```

3.2 Build-List+ solves the linearization problem

In this section, we prove the following theorem:

▶ **Theorem 3.** Build-List+ *is a self-stabilizing solution to the linearization problem.*

We prove the theorem in three steps: First, we show that starting from any initial state in which NG is weakly connected, NG will always be weakly connected. Second, we show that starting from any initial state, there will be a state in which ENG will be a supergraph of the line graph and that the explicit edges corresponding to the line will never be removed. Third, we prove that all superfluous explicit edges will eventually vanish. Note that all proofs that are omitted in this section can be found in the full version [16].

The first step is represented by the following lemma:

▶ **Lemma 4.** *If a computation of* Build-List+ *starts from a state where NG is weakly connected then in every state, NG remains weakly connected.*

For the second step of the proof of the theorem, we introduce the notation $\mathit{nextLeft}(u) := \mathit{argmax}\{id(v)|v \in \mathit{Left}(u)\}$ and $\mathit{nextRight}(u) := \mathit{argmin}\{id(v)|v \in \mathit{Right}(u)\}$. Furthermore, let $\mathit{length}(u, v)$ for two nodes u and v denote the hop distance in the (ideal) line topology between u and v. We define $rv(v)$ for a node v as $\mathit{length}(v, \mathit{nextRight}(v))$ if $\mathit{Right}(v) \neq \emptyset$ or as n if $\mathit{Right}(v) = \emptyset$; we define $lv(v)$ analogously for $\mathit{nextLeft}(v)$. With this, we define a potential function $\Phi := \sum_{i=1}^{n-1} rv(v_i) + \sum_{i=2}^{n} lv(v_i)$ where $v_1 < v_2 < \cdots < v_n$ are all nodes ordered by their id increasingly. Notice that Φ is bounded from above by $2n(n-1)$ and from below by $2(n-1)$. Also notice that according to the protocol, $\mathit{nextLeft}(v)$ ($\mathit{nextRight}(v)$) can only change if v puts a node closer to v than $\mathit{nextLeft}(v)$ ($\mathit{nextRight}(v)$) into Left (Right). Thus, Φ never increases. We define the *closest neighbor graph* as the graph $G_{NB} = (V, E_{NB})$ where V is the set of all nodes and $(x, y) \in E_{NB}$ iff $y = \mathit{nextRight}(x) \vee y = \mathit{nextLeft}(x)$. Furthermore, we say an edge is *temporary* if it is an implicit edge due to a TempDelegate() message. All other types of implicit edges are called *non-temporary*. One can show the following:

▶ **Lemma 5.** *Assume there is a system state such that Φ does not decrease in any further step of the computation. Then G_{NB} is bidirected and strongly connected.*

We prove this lemma step-by-step, starting with the following lemma:

▶ **Lemma 6.** *Assume a system state such that Φ does not decrease in any further step of the computation. Then G_{NB} is bidirected.*

The definition of a closest neighbor graph and Lemma 2 imply the following:

▶ **Corollary 7.** *If G_{NB} is bidirected and disconnected, every connected component forms a line.*

To show that G_{NB} is also strongly connected, we need two additional lemmata. We start with the following:

▶ **Lemma 8.** *Assume that in a state of the computation of* Build-List+ *G_{NB} is bidirected and disconnected. If there is a non-temporary edge (w, v) with $w \in C_1, v \notin C_1$ for a connected component C_1, then eventually either there will be an explicit or a temporary edge (x, y) with $x \in C_1$ and $y \notin C_1$ or Φ will decrease.*

▶ **Lemma 9.** *Assume that in a state of the computation of* Build-List+ G_{NB} *is bidirected and disconnected. If there is an explicit or a temporary edge* (w,v) *with* $w \in C_1$ *and* $v \notin C_1$ *for a connected component* C_1*, then eventually there will be an explicit or temporary edge* (x,y) *with* $x \in C_1, y \notin C_1$ *and* $length(x,y) < length(w,v)$*, or* Φ *will decrease.*

We are now ready to prove Lemma 5:

Proof. Assume there is an initial state in which Φ does not decrease anymore. Furthermore, assume that the closest neighbor graph G_{NB} is disconnected. Firstly, Lemma 6 guarantees that G_{NB} is bidirected. Furthermore, by Lemma 4, there must be at least one (implicit or explicit) edge (w,v) between a connected component C_1 and another connected component. Together with Lemma 8 this implies that at some point there must be a temporary or explicit edge (x,y) with $x \in C_1$ and $y \notin C_1$. However, then Lemma 9 can be applied. Since there is only a finite number of times that there can be a shorter edge, at some state, Φ must decrease, yielding a contradiction. Thus G_{NB} must be weakly connected. Note that Lemma 6 implies that G_{NB} is also strongly connected, yielding the claim of Lemma 5. ◀

Note that since Φ can never increase and since Φ is bounded from below, Φ can only decrease for a finite number of states. After that, the conditions of Lemma 5 are fulfilled. This lemma and Corollary 7 imply the following corollary:

▶ **Corollary 10.** *For any computation of* Build-List+*, there is a state in which the graph formed by the explicit edges is a supergraph of the line topology.*

For the third step of the proof of the theorem, we have the following lemma:

▶ **Lemma 11.** *If a computation of* Build-List+ *contains a state in which* ENG *is a supergraph of the line topology, then there will be a suffix in which* ENG *is the line topology and no new explicit edges will ever be created again.*

Note that Corollary 10 and Lemma 11 imply that Build-List+ converges to the list. Moreover, Lemma 11 yields the closure property. This finishes the proof of Theorem 3.

3.3 Build-List+ satisfies non-trivial monotonic searchability

In this subsection we prove the following theorem:

▶ **Theorem 12.** Build-List+ *admissible-message satisfies non-trivial monotonic searchability according to* Search+*.*

Note that all proofs that are omitted in this section can be found in the full version [16].

We start with some preliminaries. First we define $R(v)$ as the set of all nodes x with $id(v) < id(x)$ for which there is a directed path from v to x consisting solely of explicit edges (y,z) with $id(y) < id(z)$. Furthermore, we define $R(v,ID) := \{x \in R(v) | id(x) \leq ID\}$. In addition, we define $L(v)$ as the set of all nodes x with $id(x) < id(v)$ for which there is a directed path from v to x consisting solely of explicit edges (y,z) with $id(z) < id(y)$. For a set U, $R(U) := U \cup \bigcup_{u \in U} R(u)$ and $R(U,ID) := \{x \in R(U) | id(x) \leq ID\}$. Accordingly, $L(U) := U \cup \bigcup_{u \in U} L(u)$ and $L(U,ID) := \{x \in L(U) | id(x) \geq ID\}$.

Moreover, we define a state as admissible if the following message invariants hold:

1. If there is an Introduce(v,w) message with $w \neq \perp$ in $u.Ch$, then $v \neq w$, and $u \in R(w)$ (or $u \in L(w)$).
2. If there is a Linearize(v) message in $w.Ch$, then there is a node $u \neq v$ with $u \in Right(w)$ and $v \in R(u)$ if $w < v$ (or $u \in Left(w)$ and $v \in L(u)$ if $v < w$).

3. If there is a ForwardProbe$(source, destID, Next, seq)$ message in $u.Ch$, then
   a. $id(source) < destID$ and $\forall x \in Next : id(x) \geq id(u)$ and $u = argmin_u\{id(u)|u \in Next\}$ (alternatively $destID < id(source)$ and $\forall x \in Next : id(x) \leq id(u)$ and $u = argmax_u\{id(u)|u \in Next\}$).
   b. $id(source) < destID$ and $R(next) \subseteq R(source)$ (or $destID < id(source)$ and $u \in L(source)$).
   c. if v exists such that $id(v) = destID$ and $id(source) < destID$ and $v \notin R(Next, destID)$ (or $id(source) < destID$ and $v \notin L(Next, destID)$) then for every admissible state with $source.seq[destID] < seq$, $v \notin R(source, destID)$ ($v \notin L(source, destID)$).
4. If there is a ProbeSuccess$(destID, seq, dest)$ message in $u.Ch$, then $id(dest) = destID$ and $dest \in R(u)$ if $destID > id(u)$ (or $dest \in L(u)$ if $destID < id(u)$).
5. If there is a ProbeFail$(destID, seq)$ message in $u.Ch$, then either there is no node with id $destID$, or for every admissible state with $u.seq[destID] < seq$, $v \notin R(u)$ (and $v \notin L(u)$), where v such that $id(v) = destID$.
6. If there is a Search$(v, destID)$ message in $u.Ch$, then $id(u) = destID$ and $u \in R(v)$ if $id(v) < destID$ (or $u \in L(v)$ if $destID < id(v)$).

One can show the following Lemma 13 and Lemma 14 which together imply Corollary 15.

▶ **Lemma 13.** *If in a computation of* Build-List+*, there is an admissible state, then all subsequent states are admissible.*

▶ **Lemma 14.** *In every computation of* Build-List+ *there is an admissible state.*

▶ **Corollary 15.** *In every computation of* Build-List+*, there exists a suffix in which every state is admissible.*

For the rest of this subsection, we assume that every computation starts in an admissible state, since we want to show monotonic searchability must hold starting from admissible states only. Furthermore, w.l.o.g., we only consider Search$(u, destID)$ messages with $id(u) < destID$.

Before we can prove Theorem 12, we need an additional result:

▶ **Lemma 16.** *For every message $m =$ ForwardProbe$(v, destID, Next, seq) \in u.Ch$ with $id(u) < destID$, it holds that if there is a node w with $id(w) = destID$ and $w \in R(u)$, then there will be a state with $m' =$ ForwardProbe$(v, destID, Next', seq) \in w.Ch$.*

We are now ready to prove Theorem 12:

Proof. Let m, m' be two Search$(u, destID)$ messages initiated in u in admissible states with m being initiated before m' and assume that m is delivered successfully, but m' is not. Let v be such that $id(v) = destID$. Note that if m' is added to the set $WaitingFor[destID]$ when m is already in the set, then the protocol will handle both messages identical, i.e., if m is successfully delivered to v due to an ProbeSuccess() message, m' is as well. Therefore, m' is added to $WaitingFor[destID]$ when $m \notin WaitingFor[destID]$, which implies $u.seq[destID]$ has increased since the successful delivery of m (according to the protocol). Since we assume that m' is not delivered successfully, either a ProbeFail$(dest, seq)$ message eventually arrives at u with $seq \geq u.s[destID]$, or no ProbeSuccess$(destID, seq, dest)$ with $seq \geq u.s[destID]$, $dest = destID$ will ever arrive at u. We consider both cases individually. In the first case, by the fifth invariant, $v \notin R(u)$ has to hold even though m was already successfully delivered. By the sixth invariant, when m was delivered, $v \in R(u)$, which is why this is a contradiction to Lemma **??**. In the second case, note that ForwardProbe$(u, destID, \{u\}, seq)$ messages are regularly initiated by u with $seq \geq$

$u.s[destID]$ (since $u.seq$ is monotonically increasing). Again, due to the successful delivery of m, by the sixth invariant and Lemma **??**, $v \in R(u)$ when m' was initiated, and therefore, by Lemma 16, a ForwardProbe($u, destID, Next', seq$) message with $seq \geq u.s[destID]$ will eventually be in $v.Ch$, which will be answered with a ProbeSuccess($destID, seq, v$) message, causing m' to be sent to v. By the fair message receipt assumption, this contradicts the assumption that m' is not successfully delivered. ◀

4 The Build-List* and the Search* protocols

For the Build-List+ protocol in Section 3 we implicitly assumed a static node set, i.e., nodes are not allowed to leave or join the network. In this section we want investigate monotonic searchability in terms of the *Finite Departure Problem* ($\mathcal{FDP}$) of [7]. Naturally, a leaving node does not execute InitiateNewSearch(), since it aims at leaving the system. Additionally, a leaving node that is the destination of a ForwardProbe() message, will deliberately answer with ProbeFail(). Consequently, monotonic searchability can only be maintained for pairs of staying nodes.

We note that the $\mathcal{FDP}$ deliberately ignores that new nodes can join the network. However, this abstraction is justified in a self-stabilizing setting, since from an algorithmic point of view for some node u a new node joining the network is the same as getting a message from a node that it has never been in contact with.

In this section, we present the Build-List* and the Search* protocols. In the full version [16], we further show that Build-List* solves the $\mathcal{FDP}$ and also the linearization problem, and extend the proofs of Section 3.3 to show that Build-List* also satisfies non-trivial monotonic searchability according to Search*.

4.1 Description of Build-List* and Search*

For two staying nodes that interact with each other, Build-List* is analogous to Build-List+. Therefore, we only specify the changes in case a node itself is leaving or receives a message from a leaving node. A leaving node distinguishes between two different kinds of neighbors: those that it already had before switching to the leaving mode (which are *Left* and *Right* from Build-List+) and those which it received while being leaving ($Temp_L$ and $Temp_R$). Searchability is only preserved for nodes in the former two sets.

For the ForwardProbe(), Introduce(), Linearize() and TempDelegate() actions, a leaving node u will always save nodes in $Temp_L$ and $Temp_R$ in cases where a staying node saves them in *Left* and *Right*. In its Timeout action, a leaving node u either introduces all its neighbors to each other and executes **exit** if $\mathcal{NIDEC}$ is true or it sends a ReverseAndLinearizeREQ() message to all neighbors. With this ReverseAndLinearizeREQ(dir) message u requests all neighbors to stop holding its reference. As it was shown in [7], leaving nodes should never send their own reference for a successful departure protocol. Therefore, a ReverseAndLinearizeREQ(dir) message only contains a value $dir \in \{left, right\}$ that indicates whether a left or right neighbor should be removed, i.e., u sends a ReverseAndLinearizeREQ(left) message to all its neighbors to the right and and a ReverseAndLinearizeREQ(right) message to all its neighbors to the left. If a node v receives a ReverseAndLinearizeREQ(dir) message, there are two possible scenarios. If v is staying, it sends a ReverseAndLinearizeACK(v,uniqueValue) message to all neighbors in the given direction, which contains its own reference and for each neighbor a uniquely created value (i.e., in our case a local counter or the *id* of a node would be sufficient). This values is also saved as satellite data by v at the corresponding node reference in the neighbor set.

If v is leaving, it behaves like a staying node if the *dir* is right; otherwise it ignores the request. Thereby, leaving nodes with a higher id are given a higher priority for exiting the system. Once a leaving node u receives a ReverseAndLinearizeACK(v,uniqueValue) message, it responds with ReverseAndLinearize($nodeList$, $uniqueValue$) message that contains the received unique value (for identification purposes) and also all its neighbors that are on the opposite of the node in the message (i.e., if the received node is to the right of u, u sends all left neighbors and vice-versa). A ReverseAndLinearizeACK(v,uniqueValue) message is ignored by a staying node, meaning that it is transformed into a TempDelegate(v) to itself. Finally, the ReverseAndLinearize($nodeList$, $uniqueValue$) message is received by v and v checks if it has a neighbor with the given unique value. If this is the case, v either finishes the reversal process by deleting the reference to u and saving the newly received neighbors (if v is staying or getting the ReverseAndLinearize($nodeList$, $uniqueValue$) message from a right neighbor) or v ignores the message by simply saving all nodes in $Temp_L$ (if v is leaving and getting the ReverseAndLinearize($nodeList$, $uniqueValue$) message from a left neighbor). In case the unique value does not match, the ReverseAndLinearize($nodeList$, $uniqueValue$) message is not a response to a former ReverseAndLinearizeACK(v,uniqueValue) message and all received nodes are processed by TempDelegate() messages to v itself.

The Search* protocol is very similar to the Search+ protocol. As already mentioned, leaving nodes will neither execute InitiateNewSearch(), nor will they send out a ProbeSuccess() message. In fact the only action that is different in multiple places is the ForwardProbe() action, since we have to make sure that references are not saved in *Left* and *Right* but in $Temp_L$ and $Temp_R$.

Similar to Build-List+, Build-List* performs a sanity check for $Temp_L$, $Temp_R$, *Left* and *Right* before each action. The same is done for the *nodeList* received in a ReverseAndLinearize() message. However, in the last case a failing sanity check (i.e., the nodes in *nodeList* are from two different sides of the current node) directly implies that the message is corrupt and it is safe to process the nodes with TempDelegate(). The pseudocode for Build-List* and Search* can be found in the full version [16].

We have the following results regarding Build-List*:

▶ **Theorem 17.** Build-List* *is a self-stabilizing solution to the* $\mathcal{FDP}$.

▶ **Theorem 18.** Build-List* *is a self-stabilizing solution to the linearization problem.*

▶ **Theorem 19.** Build-List* *admissible-message satisfies non-trivial monotonic searchability according to* Search*.

The proofs of these theorems can be found in the full version [16].

5 Conclusion and Outlook

To the best of our knowledge, we presented the first protocol that self-stabilizes a topology whilst satisfying monotonic searchability. We focused on the line topology as a starting point and extended our protocol such that it additionally solves the Finite Departure Problem. In the design of our protocol, it turned out that the principle of delegating explicit edges only if they have been successfully introduced before is crucial to enable monotonic searchability. A natural open question is whether the application of this principle is sufficient for monotonic searchability. That is, does applying this principle to other protocols that stabilize a topology (e.g., rings, skip-graphs, Delaunay graphs) directly yield monotonic searchability, or do other topologies require more-specialized solutions?

References

1 James Aspnes and Yinghua Wu. O(logn)-time overlay network construction from graphs with out-degree 1. In *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, pages 286–300, 2007.

2 Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. Building self-stabilizing overlay networks with the transitive closure framework. *Theor. Comput. Sci.*, 512:2–14, 2013.

3 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. `doi:10.1145/226643.226647`.

4 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

5 Shlomi Dolev and Ronen I. Kat. Hypertree for self-stabilizing peer-to-peer systems. *Distributed Computing*, 20(5):375–388, 2008.

6 Shlomi Dolev and Nir Tzachar. Spanders: Distributed spanning expanders. *Sci. Comput. Program.*, 78(5):544–555, 2013.

7 Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. On stabilizing departures in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems – 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 – October 1, 2014. Proceedings*, pages 48–62, 2014.

8 Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A note on the parallel runtime of self-stabilizing graph linearization. *Theory Comput. Syst.*, 55(1):110–135, 2014. `doi:10.1007/s00224-013-9504-x`.

9 Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Skip^{+}: A self-stabilizing skip graph. *J. ACM*, 61(6):36:1–36:26, 2014. `doi:10.1145/2629695`.

10 Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. Towards higher-dimensional topological self-stabilization: A distributed algorithm for delaunay graphs. *Theor. Comput. Sci.*, 457:137–148, 2012.

11 Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. A self-stabilization process for small-world networks. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1261–1271, 2012.

12 Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. Re-chord: A self-stabilizing chord overlay network. *Theory Comput. Syst.*, 55(3):591–612, 2014. `doi:10.1007/s00224-012-9431-2`.

13 Andreas Koutsopoulos, Christian Scheideler, and Thim Strothmann. Towards a universal approach for the finite departure problem in overlay networks. In *Stabilization, Safety, and Security of Distributed Systems – 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, pages 201–216, 2015.

14 Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. *Theor. Comput. Sci.*, 512:119–129, 2013. `doi:10.1016/j.tcs.2012.08.029`.

15 Melih Onus, Andréa W. Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*, 2007.

16 C. Scheideler, A. Setzer, and T. Strothmann. Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures (full version). *ArXiv e-prints*, December 2015. `arXiv:1512.06593`.

17 Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005), 31 August – 2 September 2005, Konstanz, Germany*, pages 39–46, 2005.

18 Yukiko Yamauchi and Sébastien Tixeuil. Monotonic stabilization. In *Principles of Distributed Systems – 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 475–490, 2010.

QuickLex: A Fast Algorithm for Consistent Global States Enumeration of Distributed Computations

Yen-Jung Chang[1] **and Vijay K. Garg**[2]

1 **Department of Electrical and Computer Engineering, University of Texas, Austin, USA**
`cyenjung@utexas.edu`

2 **Department of Electrical and Computer Engineering, University of Texas, Austin, USA**
`garg@ece.utexas.edu`

Abstract

Verifying the correctness of executions of concurrent and distributed programs is difficult because they show nondeterministic behavior due to different process scheduling order. Predicate detection can alleviate this problem by predicting whether the user-specified condition (predicate) could have become true in any global state of the given concurrent or distributed computation. The method is predictive because it generates inferred global states from the observed execution path and then checks if those global states satisfy the predicate. An important part of the predicate detection method is *global states enumeration*, which generates the consistent global states, including the inferred ones, of the given computation. Cooper and Marzullo gave the first enumeration algorithm based on a breadth first strategy (BFS). Later, many algorithms have been proposed to improve the space and time complexity. Among the existing algorithms, the Tree algorithm due to Jegou et al. has the smallest time complexity and requires $O(|P|)$ space, which is linear to the size of the computation P. In this paper, we present a fast algorithm, QuickLex, to enumerate global states in the lexical order. QuickLex requires much smaller space than $O(|P|)$. From our experiments, the Tree algorithm requires 2–10 times more memory space than QuickLex. Moreover, QuickLex is 4 times faster than Tree even though the asymptotic time complexity of QuickLex is higher than that of Tree. The reason is that the worst case time complexity of QuickLex happens only in computations that are not common in practice. Moreover, Tree is built on linked-lists and QuickLex can be implemented using integer arrays. In comparison with the existing lexical algorithm (Lex), QuickLex is 7 times faster and uses almost the same amount of memory as Lex. Finally, we implement a parallel-and-online predicate detector for concurrent programs using QuickLex, which can detect data races and violation of invariants in the programs.

1998 ACM Subject Classification D.2.4 [Software/Program Verification] Validation

Keywords and phrases consistent global state, algorithm, computation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.25

1 Introduction

The technique of predicate detection was first proposed for distributed debugging [5, 14]. Many tools also use this technique for detecting various types of bugs in concurrent systems [4, 9, 21, 16]. The problem of predicate detection is to detect if the user-specified condition (or simply *predicate*) could happen in the given concurrent or distributed computation, which is modeled as a partially ordered set (*poset*) of events where each event corresponds to an

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 25; pp. 25:1–25:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

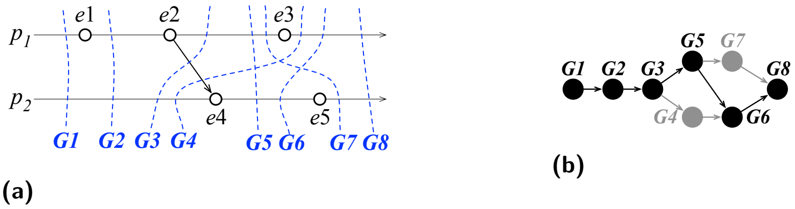


Figure 1 (a) The captured logical order between events, which form a poset. The dashed lines are consistent global states of the program. (b) The relationship of the set of consistent global states.

operation of the program. On this poset, the inferred consistent global states of the system are generated and checked if any one of them satisfies the predicate.

We use the computation in Fig. 1a to explain the technique of predicate detection. In this computation, the event $e2$, which occurs on process p_1, sends a message to event $e4$ on process p_2. Because of the message, the causal dependency between $e2$ and $e4$ is established. A global state is consistent if there exists an execution path to reach that state. In Fig. 1a, the dashed lines show all consistent global states of the computation and each global state contains all the events to the left of the corresponding dashed line. For example, the global state $G4$ contains events $e1$, $e2$, and $e3$.

Fig. 1b shows the relationship of the consistent global states in Fig. 1a. Assume that the sequence $G1, G2, G3, G5, G6, G8$ of global states is the observed execution of the program. The objective of predicate detection is to generate the inferred global states $G4$ and $G7$ without re-executing the program in order to reach $G4$ and $G7$. In this paper, we study the method for enumerating all consistent global states, including the inferred ones, of the given computation. From now on, the term *computation* refers to a concurrent or a distributed computation, the term processes refers to threads in a concurrent computation or processes in a distributed computation, and the term *global state* means consistent global state; unless specified otherwise.

Enumerating all global states of a computation P requires exponential time because the number of global states, $i(P)$, grows exponentially in n – the number of processes in the computation. With some assumptions on the predicate, the complexity may be reduced to polynomial time because only a partial set of global states is enumerated [9, 17, 23, 24, 26]. If no assumption is made regarding the predicate, i.e., the enumeration algorithm is general-purpose, then enumerating every global state is necessary. Thus, the time complexity of a general-purpose algorithm can be calculated by multiplying $i(P)$ by *the time complexity per global state*, which is the time to advance from one global state to the other. For simplicity, we use the time complexity per global state to represent the time complexity of a general-purpose algorithm.

Cooper and Marzullo [5] gave the first general-purpose enumeration algorithm based on a breadth first strategy (BFS) that requires $O(n^3)$ time and exponential space in n, which is the number of processes in the computation P. Alagar and Venkatesan [1] presented the notion of global interval which reduces the space complexity to $O(|P|)$. Steiner [28] gave an algorithm that uses $O(|P|)$ time, and Squire [27] further improved the computation time to $O(log|P|)$. Pruesse and Ruskey [25] gave an algorithm that enumerates global states in a combinatorial Gray code manner. The algorithm uses $O(|P|)$ time and can be reduced to $O(\Delta(P))$, where $\Delta(P)$ is the maximal in-degree of any event; however, the space grows exponentially in $|P|$. Later, Jegou et al. [18] and Habib et al. [15] improved the space complexity to $O(|P|)$. Ganter [10] presented an algorithm, which enumerates global states in

Table 1 Time and space complexity of algorithms.

Algorithms	Time per Global State	Space
Cooper–Marzullo [5]	$O(n^3)$	exp. in n
Alagar–Venkatesan [1]	$O(n^3)$	$O(\|P\|)$
Steiner [28]	$O(\|P\|)$	not available
Squire [27]	$O(log\|P\|)$	not available
Pruesse–Ruskey [25]	$O(\|P\|)$	exp. in $\|P\|$
Jegou and Habib et al. [18, 15]	$O(\Delta(P))$	$O(\|P\|)$
Lexical [10, 11]	$O(n^2)$	$O(1)$
QuickLex	$O(n\cdot \Delta(P))$	$O(n^2)$ †)

†) n is the number of processes in the computation P. Thus, n^2 is usually much smaller than $|P|$ because the number of events per process is much greater than n.

lexical order, and Garg [11] gave an implementation using vector clocks [8, 22]. The lexical algorithm requires $O(n^2)$ time, but the algorithm requires only $O(1)$ space besides the input, i.e., the computation. Note that the space complexity of an enumeration algorithm only considers the memory space that stores the intermediate information during the enumeration. Table 1 summarizes the time and space complexity of the algorithms.

In this paper, we present QuickLex — a fast algorithm for global states enumeration in lexical order. In comparison with the existing lexical algorithm (Lex) [10, 11], QuickLex reduces the time complexity from $O(n^2)$ to $O(n\cdot \Delta(P))$. The time complexity can be reduced to $O(n)$ for the commonly used computations [4, 20, 9, 15, 18], in which most events send and receive at most one message.

Both QuickLex and Lex algorithms enumerate global states in the same order. However, they are fundamentally different in computing the next global state in the lexical order. The Lex algorithm simply uses the current global state and vector clocks to determine the next global state. Thus, it has to repeatedly calculate the information that is reusable. QuickLex reduces the computational cost using two approaches. First, it preprocesses the computation and pre-calculates the statically reusable information. Second, it incorporates dynamic programming to reuse the dynamic information during the enumeration.

We evaluate QuickLex using multiple benchmarks including four computations that are captured from the executions of real-world applications. In our experiments, QuickLex is 7 times faster than the Lex algorithm [10, 11] and 4–5 times faster than the Tree algorithm [15, 18]. We note here that QuickLex is faster than the Tree algorithm even though the asymptotic worst case time complexity for the Tree algorithm is lower. There are two reasons for this. First, the time complexity of QuickLex is calculated as the worst case, which is not a common computation in practice. Second, the Tree algorithm needs to store its temporary spanning tree in a linked-list, which induces large overhead during enumeration; QuickLex only uses arrays. As far as space complexity is concerned, QuickLex uses almost the same amount of memory as Lex, which shows that the extra space for dynamic programming in QuickLex is quite small. The Tree algorithm uses 2–10 times more memory than QuickLex.

In [3], we have discussed a technique, named ParaMount, to decompose any lattice of global states into multiple sublattices and to enumerate those using Lex in a parallel-and-online fashion. Since Lex and QuickLex are similar, we can easily speed up ParaMount by replacing Lex with QuickLex. The experimental results show that QuickLex speeds up ParaMount by a factor of 3. The technique in [3] focuses on the high-level parallelization of the enumeration of consistent global states, which uses sequential enumeration algorithms for its subroutines. In this paper, we focus on the fast sequential enumeration algorithm.

The rest of the paper is organized as follows. Section 2 gives the model of computation.

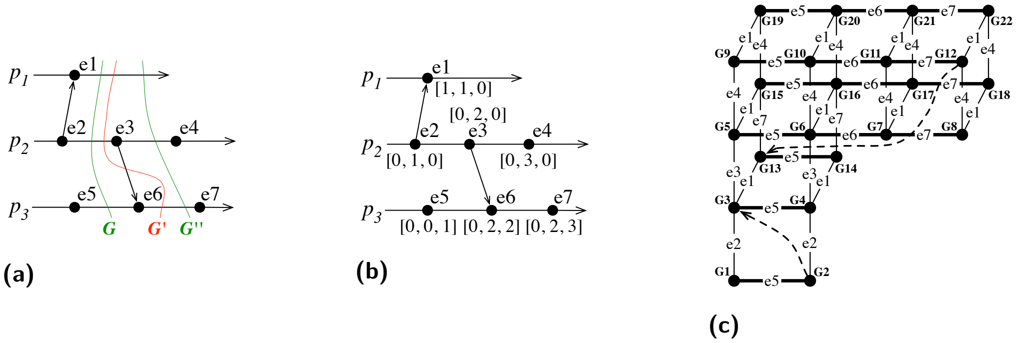


Figure 2 (a) A computation is composed of a partially ordered set (*poset*) of events. G and G'' are consistent global states and G' is an inconsistent global state. (b) The vector clocks of the events. (c) The distributive lattice formed by the set of consistent global states of the computation.

Section 3 presents the algorithm of QuickLex. Section 4 shows the experimental results. Section 5 discusses the applications of QuickLex. Section 6 concludes this paper.

2 The Model of Computations

The observed execution of the program is modeled as a computation that is composed of a poset $P = (E, \rightarrow)$ of events, which contains a set E of events together with Lamport's happened-before (HB) relation $\rightarrow$ [19]. Fig. 2a shows a graphical representation of a computation with three processes p_1, p_2, and p_3. The horizontal arrows represent the total order of the events that occur on the same process. The arrows between two events that occur on different processes represent messages. The HB relation between two events e and f is established by the following rules:

1. If e occurs before f on the same process, then $e \rightarrow f$.
2. If e sends a message and f receives the message, then $e \rightarrow f$.
3. If $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

In the computation, the HB relation between events is captured using vector clocks [8, 22]. A vector clock, vc, is an array of integers. For an event e, which occurs on process p_i, the integer $e.vc[i]$ is the index of e among the events that occur on p_i. For $j \neq i$, $e.vc[j]$ is the largest index of event f among the events that occur on process p_j such that $f \rightarrow e$. For instance, the vector clock of event $e7$ in Fig. 2b is $[0, 2, 3]$, which means the index of the current event $e7$ is 3. Moreover, the event $e3$, which has index 2 in p_2, happened before $e7$.

2.1 The Lattice of Consistent Global States

A *consistent global state* G is a subset of E, such that if G includes any event f, then it also includes all events that happened before f [2]. Formally, $G \subseteq E$ is a consistent global state if

$$\forall e, f : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G).$$

In Fig. 2a, the global states G and G'' are consistent and G' is not, because $e3 \rightarrow e6$ but $e3 \notin G'$.

A global state can equivalently be identified by the maximal events of each process. These maximal events are simply represented by an array of integers, in which the i-th integer indicates the index of the maximal event among the events that occur on process p_i. If the index is zero then no event on the corresponding process is included in the global state. For instance, G'' in Fig. 2a is represented by $[1, 2, 2]$. The symbol $G[i]$ denotes the maximal event of process p_i in G, e.g., $G''[2]$ refers to event $e3$.

The set of consistent global states forms a distributive lattice [6]. Fig. 2c shows the lattice that is formed by the consistent global states of the computation. Each node of the lattice corresponds to a consistent global state and the edge label denotes the event that takes the system from one consistent global state to the other. *The objective of QuickLex is to enumerate the lattice of consistent global states of the computation in the lexical order.*

2.2 Lexical Order among the Global States of the Computation

A lexical algorithm explores the lattice of global states using a pre-defined total order, called lexical order (denoted $\prec$), among the global states. The order $\prec$ is defined on global states as follows:

$$G \prec G' \Leftrightarrow \exists k : (\forall i : 1 \leq i < k : G[i] = G'[i]) \wedge (G[k] < G'[k]),$$

where G and G' are two arbitrary global states in the lattice. In Fig. 2c, the lexical order of the two global states $G2 = [0, 0, 1]$ and $G3 = [0, 1, 0]$ is $G2 \prec G3$. The number of each global state in Fig. 2c is its lexical order among the global states in the lattice.

2.3 Remote Events and Predecessor of an Event

If an event r sends a message to an event e, r is the remote event of e. Formally, an event r is a *remote* event of event e if 1) $r \rightarrow e$, 2) r and e occur on different processes, and 3) there does not exist any event f such that $r \rightarrow f \rightarrow e$. If an event does not have any remote event, it is a local event. In Fig. 2a, for example, event $e6$'s remote event is event $e3$. Similarly, event d is the *predecessor* of e if 1) $d \rightarrow e$, 2) d and e occur on the same process, and 3) there does not exist any event f such that $d \rightarrow f \rightarrow e$. In Fig. 2a, event $e6$'s predecessor is $e5$.

3 QuickLex

3.1 Overview

For simplicity, we consider the array of indices of a global state as a number and each index is a single digit of that number. Fig. 3 shows the mapping between an array of indices and a number of digits. In a global state, the processes at the left are high priority processes and those at the right are low priority processes.

To advance from one global state to the other (which is also referred as one *iteration* in this paper) in the lexical order, we use the notion of *carrying over* from arithmetic addition, in which we continuously add one to the low-order digit of a number and propagate the carry to a higher order digit that has not reached its limit. Then, all lower order digits are reset to their least value. Similarly, QuickLex contains two main parts. The first part adds the next event of the least priority process p_n into the current global state. If the next event of p_n is not available (e.g., if the limit of the digit is reached), the carry is propagated to a higher priority process, say p_k. The second part resets the maximal events of lower priority processes, i.e., $p_{(k+1)}$ to p_n.

Algorithm 1 shows the pseudo code of QuickLex, which takes as input a computation P. The least global state L and the greatest global state M of P are acquired from the computation itself and no additional calculation is needed. Take Fig. 2a for example,

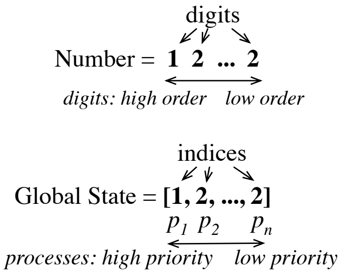


Figure 3 A number consists of multiple digits and the array of indices, which is considered as a number and each index is considered as a digit of that number.

Algorithm 1 QuickLex(P)

```
Input: A computation $P$ with $L$ as the least global state
       and $M$ as the greatest global state.
 1: $G := L$                        ▷ Use $L$ as the initial global state.
 2: for every event $e$ in $P$ do LOCATEREMOTEEVENTS($e$)
 3: INITIALSTACKS()
 4: while true do
 5:     enumerate($G$)              ▷ Evaluate the predicate on $G$.
 6:     $k :=$ PROPAGATE($G, M$)    ▷ Find $p_k$ to propagate.
 7:     if $k < 1$ then break       ▷ true: no process to propagate.
 8:     $G[k] := G[k] + 1$          ▷ Add the new event $e_k$ into $G$.
 9:     RESET($G, k$)               ▷ Reset the maximal events of lower
        priority processes, i.e., $p_{k+1}$ to $p_n$.
10: end while
```

where $L = [0, 0, 0]$ and $M = [1, 3, 3]$. QuickLex enumerates every global state G such that $L \preceq G \preceq M$. The function LOCATEREMOTEEVENTS at line 2 pre-calculates the reusable information for the PROPAGATE procedure. The function INITIALIZESTACK at line 3 initializes the memory space for dynamic programming, which speeds up the RESET procedure.

Part 1 (lines 6–8): Informally, an event is *enabled* if it can be added into the current global state G without violating the consistency of G. There might be multiple enabled events with respect to G. Since we enumerate global states in the lexical order, the PROPAGATE procedure locates the enabled event that occurs on the process that has the least priority, say p_k. If k is 0, then the next global state has exceeded the maximal global state M and hence the enumeration is terminated; otherwise, the enabled event is added into G.

Once k is decided by the PROPAGATE procedure, the processes in the computation are divided into two sets: P_h and P_l. The set P_h contains the processes whose priorities are higher or equal to process p_k, and P_l contains those whose priorities are lower than p_k. In Fig. 2a, for example, if $k = 2$, then $P_h = \{p_1, p_2\}$ and $P_l = \{p_3\}$. From now on, the symbols p_h and p_l denote an arbitrary process in P_h and P_l, respectively. Moreover, $h \leq k < l$.

Part 2 (line 9): After part 1, the maximal events for P_h are decided and fixed. Thus, we need to ensure that all the events of P_l that happened before the events of P_h are included in the next global state. We define the *maximum dependency event* of any process p_l as the event, which has the largest index among the events that occur on p_l, that has to be included in G due to the consistency of the HB relation. The procedure RESET finds the maximum dependency event for every p_l.

The details of the first and second part of QuickLex are described next.

3.2 Part 1: Procedure propagate and the Enabled Event e_k

Fig. 2 shows how part 1 works during an iteration of QuickLex. Assume that the current global state is $G2 = [0, 0, 1]$ and thus the next global state to be enumerated is $G3 = [0, 1, 0]$. The advancement from $G2$ to $G3$ is shown as a dashed arrow in Fig. 2c. First, event $e6$ is considered as the next event to be added into $G2$. However, $e6$ cannot be included in $G2$ because $e3 \rightarrow e6$ and $e3 \notin G2$, i.e., $e6$ is not enabled. Thus, the carry is propagated to p_2. Since event $e2$ is enabled, it is added to $G2$. Now, we have reached an intermediate global

Algorithm 2 Locate the set $R(e)$ of remote events for event e

```
1:  function LOCATEREMOTEEVENTS($e$)
2:      Let $d$ be $e$'s predecessor.
        ▷ Find the new HB relation on event $e$.
3:      for $i$ from 1 to $n$ except $e.pid$ do    ▷ $e.pid$ is the id of the process on which $e$ occurs.
4:          if $d.vc[i] \neq e.vc[i]$ then Add $event(i, e.vc[i])$ into $RCandidate$.
5:      end for
        ▷ Find the direct HB relation on event $e$.
6:      for every $r \in RCandidate$ do
7:          Let $r'$ be any other event in $RCandidate$.
8:          if $r.vc[r.pid]$ is larger than all $r'.vc[r.pid]$ then Add $r$ to $R(e)$.
9:      end for
10: end function
```

Algorithm 3 Procedure PROPAGATE and Function ISENABLED

```
Input: The maximal global state $M$.
Output: The process $p_k$ to propagate.
procedure PROPAGATE($G, M$)
    for $k$ from $n$ to 1 do                      ▷ From $p_n$ to $p_1$.
        if $G[k] + 1 \leq M[k]$ then              ▷ $G + e_k \preceq M$
            $e_k$ := the next event on process $p_k$.
            if ISENABLED($G, e_k$) then return $k$
        end if
    end for
    return 0                                      ▷ No process to propagate.
end procedure

Input: The next event $e_k$ on process $p_k$.
Output: Returns true if $e_k$ is enabled w.r.t. $G$.
function ISENABLED($G, e_k$)
    if $e_k$ is a local event then return true
    if $\forall r \in R(e_k)$ s.t. $r.vc[r.pid] > G[r.pid]$ then return true    ▷ $r.pid$ is the id of the process on which $r$ occurs.
    return false
end function
```

state $[0, 1, 1]$. In this example, the maximal event $G[3]$ of p_3 will be reset to 0 in the second part of QuickLex and hence $G3 = [0, 1, 0]$ is reached.

▶ **Definition 1.** An event e is enabled in a global state G iff all events that happened before e are included in G.

Assuming that event e occurs on process p_i, this condition can be determined using the property of vector clocks [8, 22]: $(e.vc[i] = G[i]+1) \wedge (\forall j \neq i : e.vc[j] \leq G[j])$. Unfortunately, it takes $O(n)$ time to compare the vector clocks in the latter part of the condition. QuickLex uses the following theorem to reduce the time complexity to $O(\Delta(P))$, where $\Delta(P)$ is the maximal number of remote events for any event:

▶ **Theorem 2.** *Let $R(e)$ be the set of remote events of event e, which occurs on process p_i, and event d be the predecessor of e, then e is enabled iff $d \in G$ and $\forall r \in R(e) : r \in G$.*

Proof (Sketch). It can be shown using the property of vector clocks. ◀

Theorem 2 reduces the computational cost of the procedure that determines whether event e is enabled by ignoring the events that transitively happened before e. For example, if event e is a local event, which does not have any remote event, then e is enabled when its predecessor is included in G. In a computation P, $\Delta(P)$ is at most $(n-1)$ because there are at most $(n-1)$ events that occur on different processes and send messages to e. If any event in P can have at most one remote event [4, 20, 9, 15, 18], then $\Delta(P)$ is $O(1)$.

Algorithm 2 uses the property of vector clocks to locate the set $R(e)$ of remote events for any event e. The function has two steps. In the first step (lines 2-5), the vector clock of e and that of e's predecessor are compared. If the i-th value (except the one for e itself) of e's vector clock is updated, then a new HB relation is established between e and *event(i,* *e.vc[i])*, which is the event, whose index is $e.vc[i]$, that occurs on process p_i. However, we are interested in only direct HB relation because of Theorem 2. Thus, the second step (lines 6-9) uses another property of vector clocks: if event r has not happened-before event r', then the vector clock of r' does not contain r's latest clock value, i.e., $r.vc[r.pid]$, where pid is the id of the process on which r occurs. Note that Algorithm 2 is invoked only once at the beginning of QuickLex and the calculated $R(e)$ for event e is reused during the enumeration.

Algorithm 3 shows the procedure PROPAGATE. The procedure decides which process to propagate starting from the least to the highest priority processes in order to follow the lexical order. Moreover, the event that occurs after the currently maximal event of process p_k is chosen. Thus, the predecessor of e_k is always included in G. The function ISENABLED checks if either one of the following two conditions holds to determine whether e_k is enabled: 1) e_k is a local event or 2) all remote events of e_k are included in G. If any event in the computation has at most one remote event, then ISENABLED takes constant time. If e_k is enabled, then PROPAGATE has found the process p_k and it returns k. If the process p_k does not exist, which implies that M is reached, then PROPAGATE returns 0.

3.3 Part 2: Procedure reset and the Maximum Dependency Events

The maximal events of P_l are not always reset to index 0. Assume that we are advancing from $G12 = [0, 3, 3]$ to $G13 = [1, 1, 0]$ in Fig. 2. After PROPAGATE decides $k = 1$, we reach the intermediate global state $[1, 3, 3]$. However, we cannot simply reset the global state to $[1, 0, 0]$ because it is not consistent; it includes $e1$ but does not include $e2$ even though $e2 \rightarrow e1$ (see Fig. 2a). So, the procedure RESET has to find the *maximum dependency events* of p_2 and p_3 that would satisfy the consistency of the global state.

From now on, the symbol $G_m[l]$ denotes the maximum dependency event of p_l, which becomes the maximal event $G[l]$ of p_l after RESET. When e_k is decided, the maximal events of P_h are also decided. The maximum dependency event $G_m[l]$ for every p_l can be calculated using the property of vector clocks:

$$G_m[l] = \max_{1 \leq j \leq n} (G[j].vc[l])$$

For simplicity, the expression $\max_{1 \leq j \leq i}(G[j].vc[l])$ is denoted by the symbol $X_l(i)$ from now on. Fig. 4 shows how the maximum dependency event $G_m[l]$ of a process p_l is identified by $X_l(n)$. In Fig. 4, the events $e1$, $e2$, $e3$, and $e4$ are four events that occur on process p_5. Assume that their indices are 1, 2, 3, and 4, respectively. Suppose that $k = 4$. Thus, $G[4]$ is the new event e_k. The fifth indices of the vector clocks of the maximal events of p_1, p_2, p_3, and p_4 are shown in the figure (i.e., $G[1].vc[5]$, $G[2].vc[5]$, $G[3].vc[5]$, and $G[4].vc[5]$). The bold arrows between events are the HB relations that are obtained from these indices. Since $G[3].vc[5]$ has the largest index, i.e., 4, it follows that $e4$ is the maximum dependency event of p_5. In other words, $G_m[5] = X_5(4) = 4$.

In fact, $G_m[l]$ can be identified by $X_l(k)$ instead of $X_l(n)$:

▶ **Theorem 3.** *For a global state G, k, and any process p_l, $X_l(i) = X_l(k)$ for all $i > k$.*

Proof. Assume that the condition is not true, i.e., $\exists i : i > k : X_l(i) > X_l(k)$. The condition implies that $G_m[l] \rightarrow e_i$, which is an event that occurs on process p_i. Because $i > k$, we get $p_i \in P_l$ and thus $e_i \rightarrow G_m[i]$; so e_i is included in G. Moreover, since $G_m[i]$ is a maximal dependency event, there exists an event e_h such that $G_m[i] \rightarrow e_h$, where e_h occurs on a process p_h, where $h \leq k$.

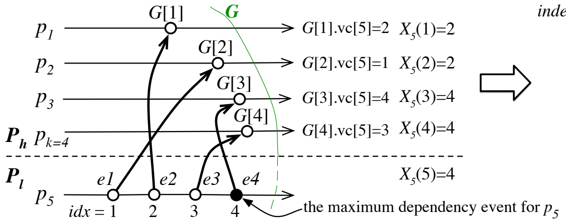


Figure 4 The symbol $X_l(i)$ denotes the function $max_{1\leq j\leq i}G[j].vc[l]$. The upside-down $stack_5$ on the right is the actual $stack_l$ that is used by QuickLex.

Algorithm 4 Incremental update of array X_l

```
Input: The process id of $p_l$, the decided $k$,
    and $\forall i : 1 \leq i \leq n : X_l[i] = X_l(i)$ w.r.t.
    global state $F$.
Output: $\forall i : 1 \leq i \leq n : X_l[i] = X_l(i)$ w.r.t.
    global state $G$.

function UPDATEARRAYX($l, k$)
    $X_l[k] := \max\big(X_l[k-1], G[k].vc[l]\big)$
    for $i$ from $(k+1)$ to $n$ do $X_l[i] := X_l[k]$
end function
```

Algorithm 5 Initialize stacks for every process

```
function INITIALIZESTACKS()
    for $i$ from 1 to $n$ do ▷ For every process $p_i$ in $P$.
        push $[p_1 : G[1].vc[i]]$ into $stack_i$
        for $j$ from 1 to $(i-1)$ do ▷ $k < i$ is always true.
            if $top.val < G[j].vc[i]$ then
                push $[p_j : G[j].vc[i]]$ into $stack_i$
        end for
    end for
end function
```

Due to the transitivity of HB relation, we get $G_m[l] \rightarrow e_i \rightarrow G_m[i] \rightarrow e_h$ and hence $X_l(h)$ also contains the largest value of $X_l(i)$. Since $h \leq k < i$, we get $X_l(h) = X_l(k) = X_l(i)$, which contradicts the assumption. ◀

According to Theorem 3, $X_l(k)$ has the largest clock value among $X_l(i)$ for all i. Consequently, $G_m[l]$ can be identified by $X_l(k)$. Now we show how to calculate the value of $X_l(k)$ in amortized constant time for each iteration using dynamic programming. It is easy to see that the value of $X_l(i)$ satisfies the following recursive equation:

$$X_l(i) = \begin{cases} G[1].vc[l], & \text{if } i = 1 \\ max\big(X_l(i-1), G[i].vc[l]\big), & \text{otherwise} \end{cases} \tag{1}$$

We use an auxiliary integer array X_l for each process p_l, in which each value $X_l[i]$ stores the value of $X_l(i)$. Note that $X_l(i)$ is the value of $\max_{1\leq j\leq i}(G[j].vc[l])$ and $X_l[i]$ is a calculated result. The array X_l has to satisfy the invariant:

$$\forall i : 1 \leq i \leq n : X_l[i] = X_l(i)$$

For any global state G and a given k, we can calculate the array X_l for each process p_l with respect to G. Assume that F is the previous global state of G in the lexical order. Instead of calculating the array X_l for G from scratch, we incrementally construct X_l from that of F. The incremental update procedure is shown in the function UPDATEARRAYX in Algorithm 4.

▶ **Theorem 4.** *Function* UPDATEARRAYX *maintains the invariant of X_l after the incremental update.*

Algorithm 6 Function UPDATESTACK and Procedure RESET

```
Input: The process id of $p_l$ and the decided $k$.
Output: The top value of $stack_l$ is $G[l]$.
 1: function UPDATESTACK($l, k$)
 2:     pop $stack_l$ until $top.pid \leq k$.
 3:     if $top.val < G[k].vc[l]$ then
 4:         if $top.pid = k$ then $top.val := G[k].vc[l]$
 5:         else push $[p_k : G[k].vc[l]]$ into $stack_l$
 6:     end if
 7: end function

Input: The decided $k$.
Output: The maximum dependency events of
        $P_l$ are found.
 8: procedure RESET($G, k$)
 9:     for $l$ from $(k + 1)$ to $n$ do
10:         UPDATESTACK($l, k$)
11:         $G[l] := top.val$          ▷ Set $G[l]$ to $G_m[l]$.
12:     end for
13: end procedure
```

Proof. We consider the three intervals of the values in X_l:

(a) $i < k$: Since the maximal events of P_h are not changed, the true values of $X_l(i)$ for $i < k$ remain the same. Thus, UPDATEARRAYX does not need to update $X_l[i]$ for $i < k$.

(b) $i = k$: $X_l[i]$ is updated at line 2 using equation (1), where the true value of $X_l(i-1)$ is obtained from $X_l[i-1]$.

(c) $i > k$: $X_l[i]$ is updated at line 3 using Theorem 3. ◀

Since the results of X_l are non-decreasing, we only need to store the values that are larger than their previous one and the process ids of the events that provide the values. For instance, $stack_5$ in Fig. 4 is the actual stack (which is shown upside down) for storing the results of X_5. In $stack_5$, the top entry $[p_3 : 4]$ means $X_5[3] = X_5[4] = \cdots = X_5[n] = 4$ and the bottom entry $[p_1 : 2]$ means $X_5[1] = X_5[2] = 2$.

Algorithm 5 constructs the $stack_i$ of each process p_i for the initial global state of a computation, which is $[0, 0, ..., 0]$. Although k does not exist in the initial global state, we know that $k < i$ for each process p_i because of the definition of P_l. Therefore, it is safe to assume that $k = (i-1)$ when constructing $stack_i$. It is easy to see that the construction of $stack_i$ is equivalent to the construction of the array X_i. Moreover, the function UPDATEARRAYX in Algorithm 4 can be converted to the function UPDATESTACK in Algorithm 6. Line 2 of UPDATEARRAYX is equivalent to lines 2-6 of UPDATESTACK and line 3 of UPDATEARRAYX is achieved by the property of $stack_l$.

▶ **Theorem 5.** *$G_m[l]$ can be identified using $stack_l$ in an amortized constant time per global state.*

Proof. At line 2 of Algorithm 6, if $stack_l$ pops m entries, then there exist m iterations that cumulatively pushed m entries into $stack_l$. Therefore, the cost of the pop operations can be evenly charged to the m iterations and be reduced to amortized constant time. The operations at lines 4 and 5 take constant time. As a result, the time complexity for updating a stack is amortized constant time per global state. ◀

Finally, lines 8-13 of Algorithm 6 shows the procedure RESET, which updates $stack_l$ for every p_l. The maximum dependency event of p_l is identified from the top entry of $stack_l$.

3.4 The Correctness and Worst Time Complexity of QuickLex

▶ **Theorem 6.** *QuickLex enumerates the lattice of global states of a computation in the lexical order such that every global state is enumerated exactly once.*

Proof. Assume that F is the previously enumerated global state and G is the current global state to be enumerated.

Table 2 The information of benchmarks and runtimes (sec.) of each algorithm.

Benchmark	n	#events	#global states	BFS	Tree	Lex	QuickLex
d-300	10	300	42,695,907	58.43	3.80	3.41	0.76
d-500	10	500	237,475,992	375.06	19.40	18.67	3.78
d-10K	10	10,000	4,962,876,973	8,211.87	393.74	448.28	86.38
bank	8	96	815,730,721	out of memory	56.67	64.37	9.69
tsp	8	105,282	13,474,170	9.85	1.63	2.37	0.37
hedc	12	216	4,486,599,595	out of memory	322.04	488.22	78.34
elevator	12	38,528	27,643,588,608	out of memory	2,248.39	4,677.12	660.40
w-4	4	480	9,381,251	2.51	0.88	0.38	0.16
w-8	8	480	7,392,009,768	out of memory	609.74	454.28	128.03
w-12	12	480	206,379,406,870	out of memory	19,225.98	21,303.66	3,996.17
w-16	16	480	991,493,848,554	out of memory	111,452.52	179,844.62	23,263.05

Lexical Order: Since PROPAGATE adds a new event e_k to F, we get $\exists k : (\exists i : 1 \leq i < k : F[i] = G[i]) \wedge (F[k] < G[k])$ and hence $F \prec G$.

Exactly Once: Since $F \prec G$, every global state is enumerated at most once. We next show that every global state is enumerated at least once. Since $F \prec G$, we get $\forall i : 1 \leq i < k : F[i] = G[i]$ and $G[k] = F[k] + 1$. Assume that F' is a consistent global state such that $F \prec F' \prec G$. We consider the following cases:

(a) $F'[k] < F[k]$: This case implies that $F' \prec F$, which contradicts the assumption $F \prec F'$.

(b) $F'[k] = F[k]$: Since $F \prec F'$, this case implies that there exists a process $p_{k'}$ such that $k' > k$ and $p_{k'}$ has an enabled event w.r.t. F. However, PROPAGATE locates the enabled event from p_n to p_1 and hence $k' \leq k$. A contradiction.

(c) $F'[k] = F[k] + 1 = G[k]$: After RESET, any p_l cannot have a maximal event that is smaller than its maximum dependency event $G_m[l]$ due to the consistency of the HB relation. Thus, we get $\nexists l : F'[l] < G[l] = G_m[l]$. So, F' does not exist.

(d) $F'[k] > F[k]+1 = G[k]$: This case implies that $G \prec F'$, which contradicts the assumption $F' \prec G$.

◀

▶ **Theorem 7.** *The worst case time complexity of QuickLex is $O(n \cdot \Delta(P))$ per global state.*

Proof. There are two main procedures during each iteration of QuickLex: PROPAGATE and RESET. We first analyze the worst time complexity of PROPAGATE. Each invocation the function ISENABLED takes $O(\Delta(P))$ time and the *for* loop of PROPAGATE is executed at most n iterations. So, the worst time complexity of PROPAGATE is $O(n \cdot \Delta(P))$ time.

We now analyze the worst case time complexity of RESET. Each invocation of the function UPDATESTACK takes amortized $O(1)$ time and the *for* loop of RESET is executed at most n iterations. So, the worst case time complexity of RESET is amortized $O(n)$ time. As a result, the worst time complexity of each iteration of QuickLex is $O(n \cdot \Delta(P))$. ◀

4 Evaluation

4.1 Setup of Benchmarks

Table 2 shows the information of the benchmarks that are used in the experiments. The benchmarks contain three different sets of computations. The benchmarks that start with the prefix "*d-*" are randomly generated posets of events for modeling distributed computations. The benchmarks *bank*, *tsp*, *hedc*, and *elevator* are the computations that are captured from the executions of real-world concurrent applications. We establish the HB relation in these concurrent computations using the following rules [9, 20]:

1. If e occurs before f on the same thread, then $e \rightarrow f$.
2. If event e corresponds to a thread releasing a lock and f corresponds to subsequent acquisition of that lock (including implicit locks and monitors), then $e \rightarrow f$.
3. If the parent thread forks a new thread on event e and the child thread is created on event f, then $e \rightarrow f$. Similarly, if a child thread terminates on event e and the parent thread joins the child thread on event f, then $e \rightarrow f$.
4. If $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

The benchmark *banking* contains a typical error pattern in concurrent programs [7]; *tsp* is a parallel solver for the traveling salesman problem; *hedc* is a crawler for searching Internet archives; and *elevator* is a discrete event simulator for an elevator system. The benchmarks *tsp*, *hedc*, and *elevator* are the benchmark programs that are used in [4, 9, 29].

Finally, the benchmarks that start with the prefix "*w*-" have the same number of events, i.e., 480 events, but different number of processes in the computation. The set of benchmarks is used to show how different n influences the performance of enumeration algorithms, and therefore we keep the number of events constant.

4.2 Compared Enumeration Algorithms

Besides QuickLex, we implemented the breadth-first strategy (BFS) algorithm [5, 11], the ideal tree traversal algorithm (Tree) [18, 15], and the original lexical algorithm (Lex) [10, 11]. In BFS algorithm [5], a global state might be enumerated more than once, so we use the strategy in [11] to ensure that every global state is enumerated exactly once. In our experiments, we use the improved BFS algorithm.

For Lex [11], we improve the nested *for* loops of function LeastGlobalState(). Each of the *for* loop goes through process p_1 to process p_n, which takes $O(n^2)$ time. However, looping through all processes is not necessary. We modify the first loop, which only loops from p_1 to p_k, and the second loop, which only loops from p_{k+1} to p_n. Although the time complexity remains the same, the practical runtime is improved significantly. In our experiments, we use the improved Lex algorithm.

The Tree algorithm [15, 18] finds a backward spanning tree in the lattice of global states, where the root is the global state that contains all events, e.g., the state $G22$ that is shown in Fig. 2c. Then it traverses the spanning three in a depth-first manner. The performance of Tree mainly dependents on *SList* [18], which is a customized linked list that continuously adds and removes the nodes of the spanning tree. So, we use the following implementation techniques to improve its performance. First, we calculate the least number of nodes that is required by *SList* during the enumeration. Then, we pre-allocate all the nodes in an object pool, which is implemented using an array, and reuse the nodes through the enumeration procedure. Second, each node of *SList* has a counter that has to be updated and there are $\Delta(P)$ nodes that need to be updated in each iteration. We replace the counter with a timestamp, which achieves the same functionality but only needs to be set once and requires no further updates. Hence, the cost of the update is reduced from $O(\Delta(P))$ time to constant time. From our empirical observations, the implementation enhancements have reduced approximately 50

4.3 Experimental Results

The input of the compared algorithms is the vector clocks of the events in the computation and the output is the set of global states of the computation. Table 2 also shows the experimental results. All the experiments are conducted on a Linux machine with an Intel

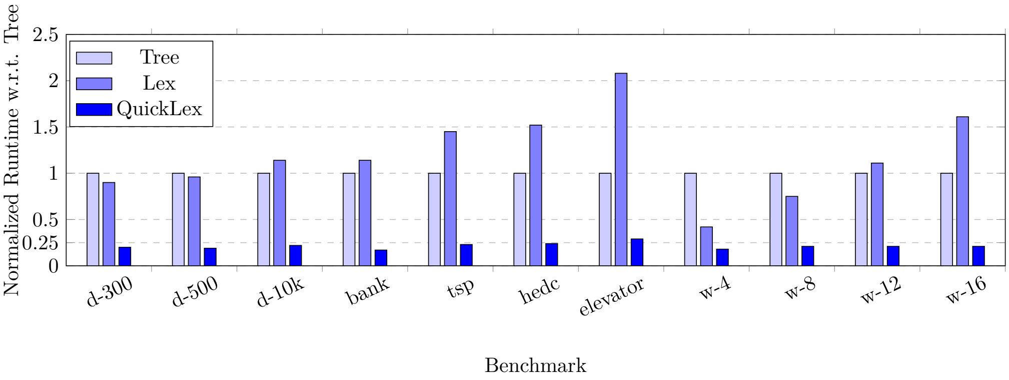


Figure 5 Normalized runtime of each algorithm w.r.t. the runtime of Tree algorithm.

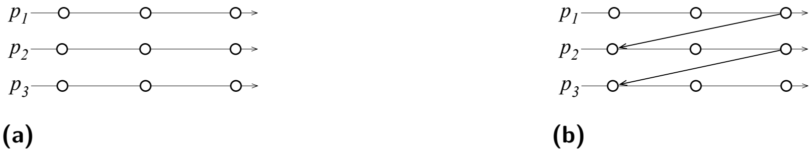


Figure 6 (a) The best case for QuickLex. (b) The worst case for QuickLex.

Xeon 2.67GHz CPU and the heap size of Java virtual machine is limited to 2GB. The runtime is measured in seconds. As it can be seen, BFS algorithm has the worst performance because of its high time complexity. Moreover, it failed to finish on more than half of the benchmarks because it ran out of the available 2GB memory. The reason is that it has to store intermediate global states for future iterations and the number of intermediate global states might grow exponentially in n in the worst case.

We first compare the runtimes of Tree, Lex, and QuickLex in the first and second set of benchmarks. Fig. 5 shows the normalized runtimes of each algorithm with respect to the runtime of Tree. We normalized the runtimes to those of Tree because it has an amortized time complexity of $O(1)$ per global state and the smallest theoretical time complexity among the existing enumeration algorithms. From Fig. 5, QuickLex is approximately 7 times faster than Lex and consistently 4–5 times faster than Tree. One reason that Tree is not as fast as QuickLex is that its intermediate information has to be stored in a linked list and therefore the cost of accessing the information is high.

We compare the runtimes of Tree, Lex, and QuickLex in the third set of benchmarks, which starts with the prefix "w-". From Fig. 5, we can see that the normalized runtimes of Lex increase as the number of processes increases. On the other hand, the normalized runtimes of QuickLex are consistently 4 times faster than those of Tree, which shows that the time complexity of QuickLex can achieve amortized $O(1)$ per global state in practice.

We now explain how QuickLex achieves amortized $O(1)$ time per global state in practice. Suppose that any event in the computation can have at most one remote event, then the worst time complexity of PROPAGATE is $O(n)$ per global state. Recall that each call of PROPAGATE runs through $(n-k+1)$ processes before returning k. If there exist more than $(n-k+1)$ global states between current and most recent PROPAGATE call that returns the same k, then the cost of current PROPAGATE call can be charged to the iterations between these two PROPAGATE calls, which cumulatively enumerated $(n-k+1)$ global states. Thus, the current PROPAGATE call is amortized to $O(1)$.

Fig. 6a illustrates the explanation. Assume that the cost of a PROPAGATE call is c if

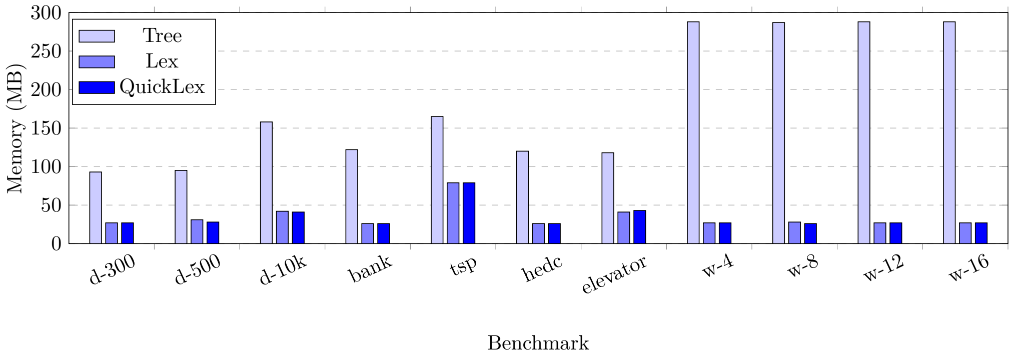


Figure 7 Memory usage of Tree, Lex, and QuickLex algorithm.

Table 3 The performance of ParaMount with different enumeration algorithms.

	Information			Runtime (ms)		# Detection
Benchmark	**LoC**	**Thread**	**#Var**	**Lex**	**QuickLex**	
banking	139	4	7	72	20	1
set (faulty)	223	4	10	152	69	1
set (correct)	260	4	10	110	51	0
arraylist1	1,474	4	6	19	19	3
arraylist2	1,377	4	16	22	15	0
sor	255	4	20	81	25	0
elevator	547	4	23	890	667	0
tsp	702	4	36	114	42	1
raytracer	1,885	4	77	1240	236	1
hedc	25,027	8	345	940	335	4

the *while* loop of PROPAGATE executes c iterations. For instance, the cost of a PROPAGATE call that returns $k = 2$ is 2. However, QuickLex has enumerated 4 global states, e.g., $[0, 0, 0]$, $[0, 0, 1]$, $[0, 0, 2]$, and $[0, 0, 3]$, between any two PROPAGATE calls that return $k = 2$. Consequently, the additional cost of the current PROPAGATE call, which returns $k = 2$, can be evenly charged to 5 global states, including the current one. Similarly, there are 17 global states for any PROPAGATE call that returns $k = 1$ to share the additional cost. As a result, the time complexity of any PROPAGATE call can be amortized to $O(1)$ time per global state. The same reason holds for the time complexity of RESET. Fig. 6b shows the worst case for QuickLex, in which only one global state exists between PROPAGATE calls. Therefore, the cost cannot be amortized and hence PROPAGATE takes $O(n)$ time. The events in this computation are totally ordered, which is not a common computation.

Fig. 7 shows the memory usage of the compared enumeration algorithms. Since Lex is stateless, its memory is mainly used for storing the input, i.e., the computation. From Fig. 7, QuickLex uses almost the same amount of memory even though QuickLex requires additional $O(n^2)$ space to store the stacks for dynamic programming. The $O(n^2)$ space is quite small because the space only stores integers. Tree, however, consumes much more memory space than Lex and QuickLex because it needs to store the information regarding its backward spanning tree, whose size is linear to $O(|P|)$. Note that $|P|$ is much larger than n^2.

5 The Applications of QuickLex

5.1 Predicate Detection in Concurrent Systems

In [3], we implemented a predicate detector, named ParaMount, for concurrent programs. ParaMount uses a sequential enumeration algorithm (e.g., Lex or QuickLex) as a subroutine to enumerate the set of global states in an online-and-parallel fashion. During the enumeration, each global state is checked for the predicate corresponding to data races. Table 3 shows the result of the detection. The columns “LoC”, “Thread”, and “#Var” show the lines of code of each benchmark, the number of threads that are used to drive each benchmark, and the number of variables of each benchmark, respectively. The column “Lex” shows the original execution time of ParaMount using the Lex as its subroutine and column “QuickLex” shows the improved execution time. On average, QuickLex improves the execution time of ParaMount by a factor of 3. The column “#Detection” shows the number of variables that have data races; all the detected variables are also detected by [4, 9].

5.2 Other Applications

In [12, 13], it has been shown that many families of combinatorial objects can be mapped to the lattice of global states of appropriate posets. Thus, lexical traversal that is discussed in this paper can also be used to efficiently enumerate all subsets of $[n]$, all subsets of $[n]$ of size m, all permutations, all permutations with a given inversion number, all integer partitions less than a given partition, all integer partitions of a given number, and all n-tuples of a product space.

6 Conclusion

In this paper, we presented a fast algorithm, named QuickLex, for global states enumeration of concurrent and distributed computations. In comparison with the original lexical algorithm, QuickLex has a preprocessing procedure and incorporates dynamic programming to reduce the time complexity from $O(n^2)$ to $O(n \cdot \Delta(P))$. In the evaluation section, we implemented and compared QuickLex with several existing enumeration algorithms, i.e., BFS [5, 11], Lex [10, 11], and Tree [18, 15]. Moreover, these algorithms are enhanced with different techniques. From our experimental results, QuickLex is 7 times faster than Lex and 4–5 times faster than Tree. The experiments also show that QuickLex can achieve amortized constant time for a certain type of computations. QuickLex uses almost the same amount of memory as Lex while Tree requires 2–10 times more memory than QuickLex. For the real-world applications, QuickLex is used to implement an online-and-parallel predicate detector for concurrent programs. The experimental results show that the detector speeds up 3 times in comparison with its previous version (which uses Lex as its subroutine).

References

1 Sridhar Alagar and Subbarayan Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Transactions on Software Engineering*, 27:412–417, 2001.

2 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.

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*, 2015.

4 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.

5 R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

6 B. A. Davey and H. A. Priestley. Introduction to lattices and order. In *Cambridge University Press*, Cambridge, UK, 1990.

7 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.

8 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.

9 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.

10 Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

11 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.

12 Vijay K. Garg. Algorithmic combinatorics based on slicing posets. *Theoretical Computer Science*, 359(1-3):200–213, 2006. doi:10.1016/j.tcs.2006.03.005.

13 Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications.* John Wiley & Sons, Inc., 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 Applied Mathematics*, 110(2-3):169–187, 2001.

16 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.

17 M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering*, 24:664–677, 1998.

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 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

20 Y. Lei and R.H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.

21 Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

22 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.

23 Vinit A. Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of International Symposium in Distributed Computing*, pages 420–434, 2007.

24 Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

25 Gara Pruesse and Frank Ruskey. Gray codes from antimatroids. *Springer LNCS Order*, 10:239–252, 1993.

26 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.

27 Matthew B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

28 George Steiner. An algorithm to generate the ideals of a partial order. *Operations Research Letters*, 5(6):317–320, 1986.

29 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.

The Synchronization Power of Atomic Bitwise Operations

Damien Imbs

Department of Mathematics, University of Bremen, Bremen, Germany
imbs@math.uni-bremen.de

Abstract

In a distributed system, processes must reach a certain level of synchronization to solve a common problem. The strongest form of synchronization can be reached through consensus: all the processes must agree on a common value that has been proposed by one of them. Consensus is universal in shared memory systems: any type of shared object can be implemented using it. Unfortunately, consensus is impossible to solve using only shared registers when processes can crash.

To circumvent this impossibility, one can use stronger objects, for example Test&Set or Compare&Swap. The synchronization power of these objects can be measured using the concept of Consensus Number: the maximum number of processes for which they can solve consensus in a crash-prone system.

Bitwise AND, OR and XOR operations are very widely used, but have received little attention in the distributed setting. Because bitwise operations are available in most modern processors, they can constitute a valuable tool for synchronization in distributed systems. It is then natural to consider the level of synchronization that these operations can achieve.

This paper introduces shared AND/OR and AND/OR/XOR registers. A shared AND/OR register consists of an array of x bits and offers three atomic operations: AND and OR operations, which take an array of x bits as parameter and change the state of the register by applying the corresponding bitwise operation, and a read operation which returns the content of the array. A shared AND/OR/XOR register additionally offers a XOR operation.

We show that shared AND/OR registers of x bits have consensus number $\lfloor \frac{x+1}{2} \rfloor$, by presenting an algorithm that solves consensus using these registers, and by proving that consensus cannot be solved for n processes using AND/OR registers that have strictly less than $2n-1$ bits. We then show that shared AND/OR/XOR registers of x bits have consensus number x using a similar technique.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Asynchronous systems, Binary operations, Consensus, Consensus number, Read/write shared memory, Shared objects, Synchronization, Wait-freedom

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.26

1 Introduction

A fundamental issue in distributed systems is the synchronization of various processes. The highest level of synchronization can be attained when processes can reach *consensus,* that is, when they can all agree on a value proposed by one of them. Consensus is *universal* in shared memory: using consensus, one can build any shared object that has a sequential specification [8, 9, 14].

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 26; pp. 26:1–26:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Unfortunately, in some communication models, consensus is impossible when even a single process can crash. This impossibility was first proven for message-passing systems [5], and was then extended to shared memory systems [13].

In order to implement consensus, one must then use objects stronger than shared registers, for example Compare&Swap. The synchronization power of such objects can be measured using their *consensus number:* the maximum number of processes for which they can solve consensus in a *wait-free* manner, that is, when any number of processes can crash [9]. Compare&Swap, for example, has consensus number $+\infty$: it can solve consensus for any number of processes. Test&Set, on the other hand, has a much lower synchronization power: its consensus number is only 2, meaning that it can solve consensus for 2 processes, but not for 3 processes.

The concept of consensus number was introduced in [9] but it was not clear whether this hierarchy was *robust*, that is, whether various objects of consensus number x could combine to give an object with a higher consensus number. In [12], an object is presented that, when only a single instance is used, can solve consensus for two processes but not for three processes. However, when x such objects are used, consensus can be solved for $x+1$ processes. The author of [12] refined the definition of the consensus number of an object to make this hierarchy robust: an object type X has consensus number x if, using any number of objects of type X and of shared registers, the maximum number of processes for which consensus can be solved is x. According to this definition, when combining various objects of different types that have consensus number at most x, one cannot obtain an object that has consensus number $y > x$.

Content of the paper

Bitwise operations are very widely used in the sequential setting but have received surprisingly little attention in the context of distributed computing. Because bitwise operations are available in most modern processors, they can constitute a valuable tool for synchronization in distributed systems. It is then natural to consider the level of synchronization that these operations can achieve.

This paper introduces the concepts of *shared AND/OR* and *AND/OR/XOR* registers. An x-bits shared AND/OR register consists of an array of x bits and offers three operations: $and()$, $or()$, and $read()$. The $and()$ and $or()$ operations take as a parameter an array of x bits, and apply the corresponding bitwise operation to the object. The $read()$ operation returns the content of the whole array. An shared AND/OR/XOR register offers an additional $xor()$ operation that applies the XOR bitwise operation.

- The paper first presents an algorithm that solves wait-free consensus for n processes using $(2n-1)$-bits AND/OR registers. The algorithm consists of a series of competitions between one process that tries to impose its input as the value chosen for the consensus, and the other processes that try to prevent it from doing so. The algorithm is formally proved correct.
- A modification of the previous algorithm that solves wait-free consensus for n processes using n-bits AND/OR/XOR registers is then presented.
- Two impossibility results are given. It is shown that consensus cannot be solved for n processes using read/write registers and AND/OR registers that have strictly less than $(2n-1)$ bits. This impossibility implies that the first algorithm is optimal with respect to the number of processes that can solve consensus using AND/OR registers, and thus shows that the consensus number of x-bits AND/OR registers is $\lfloor \frac{x+1}{2} \rfloor$. It is then shown that consensus cannot be solved for n processes using read/write registers and

AND/OR/XOR registers that have strictly less than n bits, thus showing that the second algorithm is also optimal and that the consensus number of n-bits AND/OR registers is x.

Related work

In [9], after introducing the concept of consensus number, its value is studied for various objects. Among other results, it is shown that Test&Set, Fetch&Add, Swap, the stack and the FIFO queue all have consensus number 2, Compare&Swap, memory-to-memory move, memory-to-memory-swap all have consensus number $+\infty$ and that m-register assignment (writing to m registers atomically) has consensus number $2m - 2$. The bitwise operations considered here differ from m-register assignment in that they can only modify bits of a single bounded register, whereas m-register assignment can modify any arbitrary set of m registers. Additionally, in the case of the XOR bitwise operation, the new values of the bits modified by the operation depend on their previous values; m-register assignment simply overwrites previous values.

Among the objects that have consensus number 2, the objects of the *Common2* class [2] have an additional property: they can all be implemented using consensus objects that can only be accessed by two processes. In [1] it is shown that the stack belongs to Common2. Whether the FIFO queue belongs to Common2 is still an open problem.

Test&Set belongs to the Common2 class. In [11], its specification is slightly modified to obtain a new object that has consensus number $+\infty$. The idea of the modification is to share the value returned by the operation.

In [7], a model of multicore architectures such as Compute Unified Device Architecture (CUDA) is presented, and its consensus number is studied.

The consensus problem has been generalized to the *set agreement* problem [4]. In the k-set agreement problem, processes must decide at most k different values that have been proposed by some process. The k-set agreement problem has been shown to be impossible to solve in shared memory when any number of processes can crash, even when $k = n - 1$ [3, 10, 15]. In [6], tasks (one-shot objects) are classified according to their ability to solve k-set agreement, for k from 1 (consensus) to $n - 1$.

Roadmap

The paper is composed of 6 sections. Section 2 presents the model, introduces the concept of AND/OR and AND/OR/XOR registers, and presents the consensus number hierarchy. Section 3 presents an algorithm that solves consensus using AND/OR registers. Section 4 presents a modification of the previous algorithm that solves consensus using AND/OR/XOR registers. Section 5 shows that these algorithms are optimal with respect to the number of processes that can solve consensus, and determines the consensus number of AND/OR and AND/OR/XOR registers. Finally, Section 6 concludes the paper.

2 Model and definitions

We consider a set Π of n processes $p_1, \ldots, p_n$. Processes are *asynchronous;* there is no assumption on their respective speeds. Moreover, any process can crash: it can stop its execution at any point in time.

In a given execution, a process that crashes is said to be *faulty*. Otherwise, it is *correct* and executes an infinite number of steps.

2.1 Communication model: shared memory, AND/OR registers and AND/OR/XOR registers

Processes communicate by reading and writing atomic registers. In addition, they can also access *shared AND/OR registers* or *shared AND/OR/XOR registers.*

An AND/OR register REG consists of an array $STATE$ of x bits and offers three operations: $and()$, $or()$, and $read()$. All these operations are atomic: they appear as being executed at a single time instant.

- *The $REG.and(array)$ operation.* The $and()$ operation takes as a parameter an array $array$ of x bits. It does not return a value.

  For each entry $y \in [1..x]$ of its array $STATE$, it executes the binary *and* operation with the corresponding entry $array[y]$ of the operation parameter. It then stores the result back in the y^{th} entry of its own array. More formally, it executes the following:

  $$\forall y \in [1..x] : STATE[y] \leftarrow STATE[y] \wedge array[y]$$

  Due to the nature of the binary AND operation, it can be reformulated as follows:

  $$\forall y \in [1..x] :\ \textbf{if } array[y] = 0 \textbf{ then } STATE[y] \leftarrow 0 \textbf{ end if}$$

- *The $REG.or(array)$ operation.* The $or()$ operation is similar to the $and()$ operation, the difference being that instead of applying the binary *and* operation, it applies the binary *or* operation. It executes the following.

  $$\forall y \in [1..x] : STATE[y] \leftarrow STATE[y] \vee array[y]$$

  Again due to the nature of the binary OR operation, it can be reformulated as follows:

  $$\forall y \in [1..x] :\ \textbf{if } array[y] = 1 \textbf{ then } STATE[y] \leftarrow 1 \textbf{ end if}$$

- *The $REG.read()$ operation.* The $read()$ operation atomically returns the content of the x-bits array of the register.

An AND/OR/XOR register REG offers an additional operation $xor()$.

- *The $REG.xor(array)$ operation.* The atomic $xor()$ operation is similar to the $and()$ and $or()$ operations. It applies the binary *xor* operation. It executes the following.

  $$\forall y \in [1..x] : STATE[y] \leftarrow STATE[y] \oplus array[y]$$

  Differently from the $and()$ and $or()$ operations, it cannot be reformulated as a set of writes: for any entry of *array* and $STATE$, the result depends on both values.

2.2 The consensus number hierarchy

The consensus number of an object is a measure of its power of synchronization in failure-prone shared memory systems [9]. It is based on the *consensus* problem.

Consensus is a one-shot problem: it offers a single operation propose() that each process can invoke only once. The propose(v) operation takes a value v as input and returns an output: a process *decides* a value if it chooses this value as its output.

The consensus problem is defined by the following properties.

- *Validity.* The decided value is a proposed value.
- *Agreement.* No two different processes decide different values.
- *Wait-free termination.* Every correct process decides.

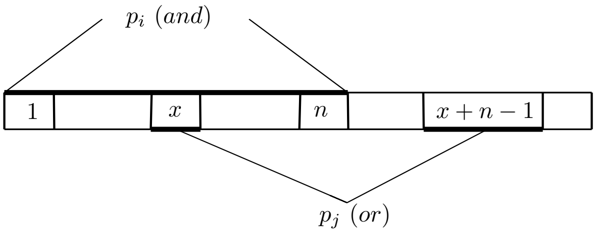


Figure 1 $AND\_OR[i]$ ($2n-1$ bits): bits modified by the owner p_i and another process p_j.

An object type T is then said to have consensus number c if, in an asynchronous shared memory system in which any number of processes can crash, consensus can be solved for c processes using atomic read/write registers and any number of objects of type T, but not for $c+1$ processes. If consensus can be solved for any number of processes, T is said to have consensus number ∞.

3 Solving consensus using AND/OR registers

In this section, we present an algorithm that solves consensus for n processes using $(2n-1)$-bits AND/OR registers. The algorithm, presented in Figure 2, constitutes a proof that x-bits AND/OR registers have consensus number at least $\lfloor \frac{x+1}{2} \rfloor$.

3.1 Mechanism of the algorithm

Every process, except p_n, competes against all the other processes. There are then $n-1$ "competitions", each associated to a single process, the "owner" of the competition. If a process wins in its competition, its value may be decided. If it loses, its value cannot be decided. Among the processes that win their competition, the one with the greatest id wins the consensus, that is, all the correct processes decide its input value. If there is no such process (all the processes $p_1, \ldots, p_{n-1}$ lost their competition), then the value of p_n is decided.

Process p_i, for $i < n$, participates in its own competition before participating in the competitions of all the other processes. Process p_n participates directly in all the competitions. If a process participates alone in its own competition, it wins. If a process participates alone in the competition of another process, the owner loses. This guarantees that, if p_n does not participate, at least one process wins its own competition and its value can be decided. If no process wins its own competition, then p_n has participated and its value can be decided.

Each competition uses an AND/OR register $AND\_OR[i]$ of $2n-1$ bits. The register is initialized with $AND\_OR[i][1..n] = [1, \ldots, 1]$ and $AND\_OR[i][n+1..2n-1] = [0, \ldots, 0]$. The owner p_i uses an *and* operation, while the other processes use an *or* operation. Process p_i overwrites (by having the corresponding bits of the parameter of its $and()$ operation set to 0) the bits $AND\_OR[i][1..n]$. Process p_j overwrites (by having the corresponding bits of the parameter of its $or()$ operation set to 1) the bits $AND\_OR[i][x]$ and $AND\_OR[i][x+n-1]$, where $x = j+1$ if $j < i$, and $x = j$ otherwise (the difference between $j < i$ and $j > i$ comes from the fact that p_i does not have dedicated $AND\_OR[i][x]$ and $AND\_OR[i][x+n-1]$ bits). The modifications of $AND\_OR[i][1]$ by p_i and of $AND\_OR[i][x+n-1]$ by p_j allow determining if the corresponding process issued its operation. The modification of $AND\_OR[i][x]$ by both p_i and p_j allows determining which process issued its operation first. Figure 1 presents the layout of the bits of an AND/OR register modified during a competition.

```
Initially:
  $\forall x \in [1..n-1]$ :
      $\forall y \in [1..n] : AND\_OR[x][y] = 1$;
      $\forall z \in [n+1..2n-1] : AND\_OR[x][z] = 0$.

Operation propose$_i(v)$:    % Code for $p_i$ %
(01)  $IN[i] \leftarrow v$;
(02)  if $(i < n)$ then                  % $and\_array$: array of $2n-1$ bits %
(03)     $and\_array \leftarrow [0, \ldots, 0]$;
(04)     for $x$ from $n+1$ to $2n-1$ do
(05)        $and\_array[x] \leftarrow 1$
(06)     end for;
(07)     $AND\_OR[i].and(and\_array)$
(08)  end if;
(09)  for $j$ from 1 to $n-1$ do
(10)     if $(j \neq i)$ then             % $or\_array$: array of $2n-1$ bits %
(11)        $or\_array \leftarrow [0, \ldots, 0]$;
(12)        if $(i < j)$ then
(13)           $or\_array[i+1] \leftarrow 1$;
(14)           $or\_array[n+i] \leftarrow 1$
(15)        else   % $i > j$ %
(16)           $or\_array[i] \leftarrow 1$;
(17)           $or\_array[n+i-1] \leftarrow 1$
(18)        end if;
(19)        $AND\_OR[j].or(or\_array)$
(20)     end if
(21)  end for;
(22)  $output \leftarrow \perp$;
(23)  for $j$ from 1 to $n-1$ do
(24)     $current \leftarrow AND\_OR[j].read()$;
(25)     if $(current[1] = 0)$ then
(26)        if $\big(\nexists x \in [2..n] : (current[x] = 0) \wedge (current[n+x-1] = 1)\big)$
(27)           then $output \leftarrow IN[j]$
(28)        end if
(29)     end if
(30)  end for;
(31)  if $(output = \perp)$ then  $output \leftarrow IN[n]$ end if;
(32)  return$(output)$.
```

Figure 2 An algorithm that solves consensus for n processes using $(2n-1)$-bits AND/OR registers (code for p_i).

3.2 Shared objects

The algorithm uses the following shared objects.

- *An array* $IN[1..n]$ *of read/write registers.* The array IN contains one entry per process. When a process starts its execution, it writes its input in the corresponding entry of the array IN (line 01). The array is used to determine the input value of the process whose value is decided (lines 27 and 31).
- *An array* $AND\_OR[1..n-1]$ *of* $(2n-1)$*-bits shared AND/OR registers.* Each process, except p_n, has an associated AND/OR register. These registers are used as arbiters. Process p_i (for $i \neq n$) uses $AND\_OR[i]$ to compete against all other processes. In this competition, p_i uses an *and* operation (line 07), while the other processes use an *or* operation (line 19). If p_i is the first to invoke an operation on $AND\_OR[i]$, it wins and its value *may* be chosen. Otherwise, another process has invoked an operation on $AND\_OR[i]$ before p_i and the value of p_i will not be chosen. After competing in all AND/OR registers, process p_i reads them all to determine which value to decide (line 24).

3.3 Process behavior

When it begins its execution, process p_i writes its input value in its entry of the array IN (line 01). If it has an associated AND/OR register $AND\_OR[i]$ (that is, if $i \neq n$), it then prepares the array that will be used as a parameter for its *and* operation on $AND\_OR[i]$ (lines 02–06). The goal of the *and* operation is (1) to let other processes determine if p_i has issued its *and* operation when they read $AND\_OR[i]$ (line 25) and (2) to let them determine if an *or* operation has been issued on $AND\_OR[i]$ before this *and* operation (line 26). The entry $and\_array[1]$ is used to signify that p_i has issued the *and* operation, while the entries $and\_array[2..n]$ are used to determine whether p_i was the first process to issue an operation on $AND\_OR[i]$.

After issuing its *and* operation on $AND\_OR[i]$ (line 07), p_i prepares the array that will be used for the *or* operations on the AND/OR registers $AND\_OR[j]$ with $j \neq i$ (lines 11–18). In each of these registers, the *or* operation may modify 2 bits: the first bit ($AND\_OR[j][i+1]$ if $i < j$, $AND\_OR[j][i]$ otherwise) is used to determine whether p_i issued its *or* operation before the *and* operation by p_j, if the latter has been issued. The second bit ($AND\_OR[j][n+i]$ if $i < j$, $AND\_OR[j][n+i-1]$ otherwise) is used to signify that p_i issued its *or* operation on $AND\_OR[j]$.

Process p_i then determines the value that it will return. For all the AND/OR registers $AND\_OR[j]$, p_i determines whether p_j won its competition, that is, whether p_j was the first process to issue an operation on $AND\_OR[j]$ (lines 25–26). If that is the case, p_i updates its estimate of the value it has to return by setting *output* to $IN[j]$ that contains p_j's input (line 27). If no process has won its competition, p_i returns p_n's input value (line 31).

3.4 Proof of the algorithm

▶ **Lemma 1.** *The decided value is a proposed value.*

Proof. The variable *output* is returned at line 32. If it is not written at line 27, during the loop at lines 23–30, it is written at line 31. When it is returned, *output* thus always contains the value of an entry of the array IN. There are then two cases.

1. The last write of *output* is at line 27.
   The write of *output* at line 27 can only happen if the first bit of the corresponding AND/OR register $AND\_OR[j]$ has been set to 0 (read of $AND\_OR[j]$ at line 24 and condition at line 25) and if the condition at line 26 is respected. This can only happen if p_j, the process to which $AND\_OR[j]$ is associated, has invoked the *and* operation on $AND\_OR[j]$ at line 07, which it does after writing its input value in $IN[j]$ at line 01. The read of $IN[j]$ at line 27 thus returns p_j's input value.
2. The last write of *output* is at line 31.
   Let us note that, except p_n (which doesn't have an associated AND/OR register), every process competes in its own associated AND/OR register (line 07) before competing in any other AND/OR register (line 19). Let us then consider the set of participating processes minus p_n, and the first operation on an AND/OR register by any of these processes. Because operations on AND/OR registers are atomic, the first operation is well defined. Let p_i be the process that issues it.
   Suppose, by way of contradiction, that p_n issued its first operation on $AND\_OR[i]$ after p_i, or not at all. By definition of p_i, when it issues its *and* operation on the AND/OR register $AND\_OR[i]$, no other process has issued an *or* operation on it yet. Note that no other *and* operation will be issued on $AND\_OR[i]$. After this operation, the condition at line 25 is respected. The condition at line 26 is also respected: before

any process issues an *or* operation, for any $x \in [n+1..2n-1]$, $AND\_OR[i][x] = 0$ (initialization of $AND\_OR[i]$). Any subsequent *or* operation by any process p_j will set an entry $AND\_OR[i][n+x-1]$ with $x \in [2..n]$ to 1, but it will also set $AND\_OR[i][x]$ to 1 (lines 13–14 if $j < i$ or lines 16–17 if $j > i$, and *or* operation at line 19).

Any process that reads $AND\_OR[i]$ (line 24) after the first operation by p_i will then observe the conditions at lines 25 and 26 as respected, and will execute line 27. It will then not execute the write of *output* at line 31, a contradiction. In case (2), process p_n must then have issued its first operation on an AND/OR register before any other process. Its write of *IN* precedes its first operation: the read of $IN[n]$ at line 31 thus returns p_n's input value, which concludes the proof of the lemma. ◀

▶ **Lemma 2.** *No two different processes decide different values.*

Proof. The proof relies on the fact that, for each AND/OR register, the result of the competition (whether the process to which it is associated has won or lost) is fixed after the first operation applied on it. All the processes apply an operation on all AND/OR registers before deciding, and thus have the same "view" of the competition.

Let us first note that, before entering the loop at lines 23–30, and thus checking the conditions at lines 25 and 26, any process applies either an *and* or an *or* operation on all AND/OR registers (*and* operation at line 07 or *or* operations at line 19 in the loop at lines 09–21). Consider the AND/OR register $AND\_OR[i]$, for any $i \in [1..n-1]$. Because operations on AND/OR registers are atomic, the first operation on an AND/OR register is well defined. There can then be two cases.

1. The first operation on $AND\_OR[i]$ is an *and* operation.

   Let us first consider the value of $AND\_OR[i][1]$. The *and* operation by process p_i (the only *and* operation issued on $AND\_OR[i]$) sets $AND\_OR[i][1]$ to 0 (lines 04–06 and *and* operation at line 07). The condition at line 25 will then be observed as respected by all the processes which read $AND\_OR[i]$ (line 23 which, for any process, happens after its *and* or *or* operation on $AND\_OR[i]$).

   Let us now consider the condition at line 26, that depends, for any $x \in [2..n]$, on the values of $AND\_OR[i][x]$ and $AND\_OR[i][n+x-1]$. Before any process issues an *or* operation on $AND\_OR[i]$, the value of $AND\_OR[i][n+x-1]$, for any $x \in [2..n]$, is equal to 0 (initialization of $AND\_OR[i]$) and thus the condition at line 26 is respected. Because any *or* operation on $AND\_OR[i]$ happens after the first and only *and* operation on it, after any such *or* operation by a process p_j, both the values of $AND\_OR[i][j+1]$ and $AND\_OR[i][n+j]$ (if $j < i$, lines 13 and 14) or of $AND\_OR[i][j]$ and $AND\_OR[i][n+j-1]$ (if $j > i$, lines 16 and 17) will be equal to 1. The condition at line 26 will then always be observed as respected.

   Any process that reads $AND\_OR[i]$ at line 24 will then observe the conditions at lines 25 and 26 as respected, and will execute line 27, overwriting the value of *output* with p_i's input value.

2. The first operation on $AND\_OR[i]$ is an *or* operation.

   Before p_i executes its *and* operation on $AND\_OR[i]$, the value of $AND\_OR[i][1]$ is equal to 1. Any process that reads $AND\_OR[i]$ (line 24) before the *and* operation by p_i will then not execute line 27.

   Let us now consider the case of a process that reads $AND\_OR[i]$ after the *and* operation by p_i. Let p_j be the first process that issued an *or* operation on $AND\_OR[i]$. By definition, this operation happened before p_i's *and* operation. The *or* operation of p_j sets the values of $AND\_OR[i][j+1]$ and $AND\_OR[i][n+j]$ (if $j < i$, lines 13–14) or $AND\_OR[i][j]$ and $AND\_OR[i][n+j-1]$ (if $j > i$, lines 16–17) to 1 . For any $x \in [2..n]$, p_i's operation then sets $AND\_OR[i][x]$ to 0 (lines 04–06 and *and* operation, line 07). Apart from p_i, p_j is the only process that can modify its corresponding entries of $AND\_OR[i]$ ($AND\_OR[i][j+1]$ and $AND\_OR[i][n+j]$ if $j < i$, lines 13–14, or $AND\_OR[i][j]$ and $AND\_OR[i][n+j-1]$ if $j > i$, lines 16–17).

Any process that reads $AND\_OR[i]$ after the *and* operation by p_i will then observe $AND\_OR[i][x]$ equal to 0 and $AND\_OR[i][n+x-1]$ equal to 1, for $x = j+1$ if $j < i$ or $x = j$ if $j > i$. It will then not execute line 27.

We then have the following: for any AND/OR register $AND\_OR[i]$, if the first operation on it is an *and* operation, every correct process will execute $output \leftarrow IN[i]$ at line 27. If the first operation on it is an *or* operation, no process will execute $output \leftarrow IN[i]$ at line 27. Every correct process will then have the same sequence of assignments in the loop at lines 23–30. For every correct process, the last assignment to *output* will then correspond to the same entry of IN. By Lemma 1, this entry, say $IN[j]$, always corresponds to the input value of p_j, which concludes the proof of the lemma. ◀

▶ **Lemma 3.** *Every correct process decides.*

Proof. The algorithm does not contain any blocking operation, and the only three loops (lines 04–06, lines 09–21 and lines 23–30) are *for* loops, which terminate after a predetermined number of iterations. Any correct process will then terminate its execution of the algorithm by deciding a value at line 32, which concludes the proof of the lemma. ◀

▶ **Theorem 4.** *The algorithm presented in Figure 2 solves consensus for n processes in the shared memory model extended with shared $(2n-1)$-bits AND/OR registers.*

Proof. The correctness of the algorithm follows from Lemma 1 (validity), Lemma 2 (agreement) and Lemma 3 (wait-free termination). ◀

The following corollary is a direct consequence of Theorem 4.

▶ **Corollary 5.** *Shared x-bits AND/OR registers have consensus number at least $\lfloor \frac{x+1}{2} \rfloor$.*

4 Solving consensus using AND/OR/XOR registers

This section presents an algorithm that solves consensus for n processes using n-bits AND/OR/XOR registers. The algorithm, presented in Figure 3, constitutes the proof that x-bits AND/OR/XOR registers have consensus number at least x.

Differences with the previous algorithm

The main difference resides in the way that the objects are used to decide the output of each competition. The previous algorithm used two bits in each AND/OR register to determine whether the process associated to the object has issued its *and* operation before another given process has issued its *or* operation. The use by the process associated to the AND/OR/XOR register of a *xor* operation allows using a single bit to determine whether this process issued its operation before another given process. This is due to the fact that *xor* operations don't work as multiple assignments: the state of each individual bit after a *xor* operation depends on its previous state, even when it is modified.

Each competition uses an AND/OR/XOR register $AND\_OR\_XOR[i]$ of n bits. The owner p_i uses an *xor* operation, while the other processes use an *or* operation. Process p_i

```
Initially:
  ∀$x \in [1..n-1]$ :
      ∀$y \in [1..n]$ : $AND\_OR\_XOR[x][y] = 0$.

Operation propose$_i(v)$:   % Code for $p_i$ %
$IN[i] \leftarrow v$;
if $(i < n)$ then                      % $xor\_array$: array of $n$ bits %
   $xor\_array \leftarrow [1, \ldots, 1]$;
   $AND\_OR\_XOR[i].xor(xor\_array)$
end if;
for $j$ from 1 to $n-1$ do
   if $(j \neq i)$ then                % $or\_array$: array of $n$ bits %
      $or\_array \leftarrow [0, \ldots, 0]$;
      if $(i < j)$ then
         $or\_array[i+1] \leftarrow 1$
      else   % $i > j$ %
         $or\_array[i] \leftarrow 1$
      end if;
      $AND\_OR\_XOR[j].or(or\_array)$
   end if
end for;
$output \leftarrow \perp$;
for $j$ from 1 to $n-1$ do
   $current \leftarrow AND\_OR\_XOR[j].read()$;
   if $(current[1] = 1)$ then
      if $(\nexists x \in [2..n] : current[x] = 0)$
         then $output \leftarrow IN[j]$
      end if
   end if
end for;
if $(output = \perp)$ then  $output \leftarrow IN[n]$ end if;
return$(output)$.
```

Figure 3 An algorithm that solves consensus for n processes using n-bits AND/OR/XOR registers (code for p_i).

modifies (by having the corresponding bits of the parameter of its $xor()$ operation set to 1) the bits $AND\_OR\_XOR[i][1..n]$. Process p_j overwrites (by having the corresponding bit of the parameter of its $or()$ operation set to 1) a single bit $AND\_OR\_XOR[i][x]$, where $x = j+1$ if $j < i$, and $x = j$ otherwise. The modification of $AND\_OR\_XOR[i][1]$ by p_i allows determining if p_i issued its operation. The modification of $AND\_OR\_XOR[i][x]$ by both p_i and p_j allows determining (a) if p_j issued its operation (in combination with $AND\_OR\_XOR[i][1]$) and (b) which process issued its operation first. The layout of the bits of an AND/OR/XOR register modified during a competition is presented in Figure 4.

▶ **Theorem 6.** *The algorithm presented in Figure 3 solves consensus for n processes in the shared memory model extended with shared n-bits AND/OR/XOR registers.*

Proof. Apart from the differences outlined previously, the algorithm is similar to the algorithm presented in Figure 2, and thus the proof is similar to the proof of Theorem 4. ◀

The following corollary is a direct consequence of Theorem 6.

▶ **Corollary 7.** *Shared x-bits AND/OR/XOR registers have consensus number at least x.*

5 Optimality of the algorithms

This section first presents a proof that x-bits shared AND/OR registers cannot solve consensus for more than $\lfloor \frac{x+1}{2} \rfloor$ processes, when any number of processes can crash. This shows that

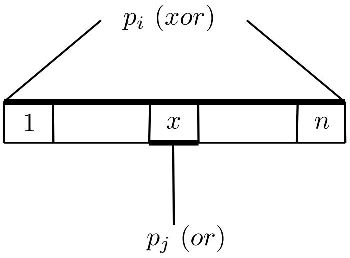


Figure 4 $AND\_OR\_XOR[i]$ (n bits): bits modified by the owner p_i and another process p_j.

the algorithm presented in Section 3 is optimal with respect to the number of processes that can solve consensus and thus that the consensus number of x-bits AND/OR registers is exactly $\lfloor \frac{x+1}{2} \rfloor$. It then presents a proof that x-bits shared AND/OR/XOR registers cannot solve consensus for more than x processes, when any number of processes can crash, and thus that the algorithm presented in Section 4 is optimal and that the consensus number of x-bits AND/OR/XOR registers is exactly x.

5.1 Preliminaries

The following concepts are used in the proofs of Lemmas 9 and 12.

Steps and executions

During a step, a process applies an atomic operation on a base shared object. An execution consists of a set of initial local states (one for each process) and a sequence of steps. A correct process takes steps in the execution until it decides (returns a value). A faulty process stops taking steps before deciding.

Prefixes and extensions

A prefix of a given execution consists of the set of initial local states of the execution and a prefix of the sequence of steps that constitutes it. An extension of a prefix starts with the prefix and is completed by a (possibly empty) sequence of steps by processes of the system. The first step that a process can take after a prefix is defined by the algorithm it is executing.

Indistinguishable executions

For a given process, an execution A is indistinguishable from an execution B if its local state is the same in both executions. Note that this is the case if the values it obtained from its operations are the same in A and in B. If it is correct and finishes its execution of the algorithm, it will then have to decide the same value in A and in B.

Valence

The prefix P of an execution is univalent if, in any execution of which P is a prefix, the correct processes decide the same value v. This means that in P, the value that processes will decide is already fixed. The same concept can apply to an extension of a prefix. A prefix that is not univalent is multivalent.

Modification of a bit

As was explained in Section 2.1, *and*() and *or*() operations can be seen as a series of assignments. The *and*() operation can only modify the bits of the object on which it is applied that correspond to the entries of its parameter that are equal to 0. The *or*() operation can only modify the bits that correspond to the entries of its parameter that are equal to 1. The *xor*() operation cannot be seen as a series of assignments, but it only modifies the bits that correspond to the entries of its parameter that are equal to 1.

In a given execution, we then say that a process can modify a given bit of a shared AND/OR or AND/OR/XOR register if its next step is (a) an *and*() operation and the corresponding bit of its parameter is set to 0, or (b) an *or*() or *xor*() operation and the corresponding bit of its parameter is set to 1.

The following lemma is similar to results that have been shown in various other papers (see e.g. [9]). It is used in the proofs of Lemmas 9 and 12. Due to page limitations, its proof is not presented here.

▶ **Lemma 8.** *Any multivalent prefix of an execution of any wait-free consensus protocol in the shared memory model extended with shared objects has a multivalent extension such that:*

1. *the next operation of any process forces a decision,*
2. *all these operations are on the same object X,*
3. *all these operations modify the state of X and*
4. *X is not a read/write register.*

5.2 Consensus number of shared AND/OR registers

▶ **Lemma 9.** *Shared AND/OR registers of x bits have consensus number at most $\lfloor \frac{x+1}{2} \rfloor$.*

Proof. Let us consider the empty prefix of an execution in which the processes propose at least two different values. By Lemma 8, there is an extension E such that E is multivalent, the next step of any process imposes a decision and these steps all modify a single AND/OR register X (and thus, they are *and* or *or* operations). We will prove that X cannot have less than $2n-1$ bits, and thus that in a system of $n > \frac{x+1}{2}$ processes in which they communicate only through read/write registers and AND/OR registers of x bits, consensus is impossible.

- *For each process p_i, there must be at least one bit of X such that only p_i can modify it.*

Consider the extension E such that, if p_i's step is the next step, then the value v_1 must be decided and, for some process p_j, if p_j's step is the next step, then the value $v_2 \neq v_1$ must be decided. Because E is multivalent, p_j must exist. Consider now the extension of E in which p_i executes one step first and then crashes, then p_j executes one step, then all other processes (if any) execute one step. Suppose now, by way of contradiction, that there is no bit of X such that only p_i can modify it. Let us recall that *and*() and *or*() operations can be viewed as assignments (Section 2.1). The *and*() and *or*() operations that follow p_i's step then overwrite its modifications to X. After these operations, the remaining processes cannot distinguish the previous extension of E, in which v_1 must be decided, from the one in which p_i did not execute its step (all other steps being executed in the same order), in which v_2 must be decided. A contradiction which proves that, for each process p_i, there must be at least one bit of X such that only p_i can modify it.

- *For each pair of processes p_i and p_j such that the next step of p_i imposes the decision of v_1 and the next step of p_j imposes the decision of $v_2 \neq v_1$, there must be at least one bit of X such that only p_i and p_j can modify it.*

Consider again the extension of E in which p_i executes one step first, then p_j, then all other processes. Suppose, again by way of contradiction, that there is no bit of X such that only p_i and p_j can modify it. All the modifications by p_i and p_j of a common bit have then been overwritten by the other processes (if p_i and p_j are the only processes in the system, $x \leq 2$ and thus, because of the previous item, they cannot modify the same bit). After all these steps, the processes cannot distinguish the previous extension of E, in which v_1 must be decided, from the one in which p_i took its step after p_j, in which v_2 must be decided. A contradiction which proves that, for each pair of processes p_i and p_j such that the next step of p_i imposes the decision of v_1 and the next step of p_j imposes the decision of $v_2 \neq v_1$, there must be at least one bit of X such that only p_i and p_j can modify it.

Let us now partition the processes in sets $P_1, \ldots, P_k$ such that, if the next operation after E is by a process in P_ℓ, the decided value has to be v_ℓ, with $\ell \neq m \Rightarrow v_\ell \neq v_m$. Because E is multivalent, there are at least two such sets and thus $k \geq 2$. The number of pairs of processes belonging to different sets is then equal to the number of edges of a complete k-partite graph in which the processes of P_ℓ correspond to the vertices of the ℓ^{th} part of the graph. For $k \geq 2$, complete k-partite graphs are connected. The minimum number of edges of a connected graph of n vertices being $n-1$, the number of pairs of processes belonging to different sets is at least $n-1$. The number of bits needed for all pairs of processes that impose different values is then at least $n-1$.

The AND/OR register X must then have at least (1) one bit per process, and (2) one bit per pair of processes that can impose different values, giving a minimum of $n+n-1 = 2n-1$. Consensus is thus impossible in a system in which processes communicate only through read/write registers and AND/OR registers of strictly less than $2n-1$ bits, and thus shared AND/OR registers of x bits have consensus number at most $\lfloor \frac{x+1}{2} \rfloor$. ◀

▶ **Theorem 10.** *Shared AND/OR registers of x bits have consensus number exactly $\lfloor \frac{x+1}{2} \rfloor$.*

Proof. The proof follows from Corollary 5 (lower bound) and Lemma 9 (upper bound). ◀

5.3 Consensus number of shared AND/OR/XOR registers

▶ **Lemma 11.** *Let S be a set and A a family of subsets of S. If $|A| > |S|$, there exists a non-empty $A' \subseteq A$ such that every element of S appears in an even number of elements of A'.*

Proof. Let $n = |S|$. Define the signature sig_B of a family B of subsets of S as an n-entries vector, in which $sig_B[i] = 0$ if the i^{th} element of S appears in an even number of elements of B, and 1 otherwise (any arbitrary total order can be used on the elements of S). There are then 2^n possible signatures for subfamilies of A.

Because $|A| > n$, there are at least 2^{n+1} different possible subfamilies of A, of which at least $2^{n+1} - 1 > 2^n$ are non-empty. By the pigeonhole principle, there must then be at least two different non-empty subfamilies B and C of A such that $sig_B = sig_C$.

Let $B' = B - (B \cap C)$ and $C' = C - (B \cap C)$. B' and C' are then disjoint. Because $sig_B = sig_C$, we also have $sig_{B'} = sig_{C'}$. Because $B \neq C$, at least one of B' and C' is non-empty.

Let $A' = B' \cup C'$. A' is then non-empty. Because B' and C' are disjoint and have the same signature, every element of S appears in an even number of elements of A', which concludes the proof of the lemma. ◀

▶ **Lemma 12.** *Shared AND/OR/XOR registers of x bits have consensus number at most x.*

Proof. Like in Lemma 9, let us consider the empty prefix of an execution in which the processes propose at least two different values. By Lemma 8, there is an extension E such that E is multivalent, the next step of any process forces a decision and these steps all modify a single AND/OR/XOR register X (and thus, they are *and*, *or* or *xor* operations). We will prove that X cannot have less than n bits, and thus that in a system of $n > x$ processes in which they communicate only through read/write registers and shared AND/OR/XOR registers of x bits, consensus is impossible.

Note that the case in which the next step of any process is an $and()$ or $or()$ operation is already covered by Lemma 9. We will thus consider that in E, the next step of at least one process is a $xor()$ operation.

Let P_{xor} be the set of processes such that their next step in E is a $xor()$ operation. Let p_i and p_j be processes in P_{xor} such that the next step of p_i in E imposes the value v_i and the next step of p_j imposes the value v_j. The execution in which p_i executes one step first (and thus v_i must be decided), then p_j executes a step, cannot be distinguished by any process from the execution in which p_j executes its step first (and thus v_j must be decided) and then p_i. All the next steps in E of processes in P_{xor} must then impose the same value v_{xor}.

- *There must be at least $|P_{xor}|$ bits of X that can only be modified by the processes in P_{xor}.*

Consider the extension of E in which first, the processes in some $P'_{xor} \subseteq P_{xor}$ execute one step (in any order) and crash. Then, a process p_i such that its next step in E imposes a value $v_i \neq v_{xor}$ executes one step, then all the remaining processes (if any) also execute one step. Suppose, by way of contradiction, that strictly less that $|P_{xor}|$ bits are modified only by the processes in P_{xor}. We will show that in this case, there exists a non-empty P'_{xor} such that this execution (in which v_{xor} must be decided) is indistinguishable by the other processes from the execution in which all the processes in P_{xor} crash before executing a step, and p_i imposes the value $v_i \neq v_{xor}$ (note that if $|P_{xor}| = 1$, no bit is modified by the single process in P_{xor} and this is trivially verified).

Let S be the set of bits modified only by the processes in P_{xor}. By Lemma 11, there exists a non-empty set $P'_{xor} \subseteq P_{xor}$ such that every bit in S is modified by an even number of processes in P'_{xor}, and thus appear unmodified after all the processes in P'_{xor} apply their operations. Because all the other bits of X are modified afterwards by $and()$ or $or()$ operations, the state of X is the same as if the processes in P'_{xor} hadn't executed their operations, a contradiction.

- *For each process p_i not in P_{xor}, there must be at least one bit of X such that the only process not in P_{xor} that can modify it is p_i.*

The reasoning is the same as in Lemma 9. Consider the execution in which first, a process $p_i \notin P_{xor}$ executes one step, imposing a value $v_i \neq v_{xor}$, and then crashes. Afterwards, a process $p_j \in P_{xor}$ executes one step, followed by all the other processes (if any) that also execute one step. If there is no bit such that the only process not in P_{xor} that can modify it is p_i, then this execution is indistinguishable by all the other processes from the execution in which p_i crashes before executing its step and in which v_{xor} is imposed, a contradiction.

There must then be at least $|P_{xor}|$ bits modified only by the processes in P_{xor} and at least $n - |P_{xor}|$ bits modified only by the processes not in P_{xor}. To solve consensus for n processes, AND/OR/XOR registers must then have at least n bits, which concludes the proof of the lemma. ◀

▶ **Theorem 13.** *Shared AND/OR/XOR registers of x bits have consensus number exactly x.*

Proof. The proof follows from Corollary 7 (lower bound) and Lemma 12 (upper bound). ◀

5.4 Other variants

Until now, we have only considered shared AND/OR and AND/OR/XOR registers. This section briefly discusses other variants. Shared AND (resp. OR or XOR) registers offer, in addition to the *read*() operation, a single operation *and*() (resp. *or*() or *xor*()). Shared OR/XOR (resp. AND/XOR) offer *or*() (resp. *and*()) and *xor*() operations, but not the *and*() (resp. *or*()) operation.

Shared AND, OR and XOR registers

Operations of a single type (*and*(), *or*() or *xor*()) are commutative: the effect of various *and*() (resp. *or*() or *xor*()) operations on the state of an object, when only these operations are issued, does not depend on the order in which they have been issued.

Consider, as in Lemmas 9 and 12, an execution E in which the next step by process p_i imposes the decision of v_i, and the next step by process p_j imposes the value $v_j \neq v_i$. Neither p_i nor p_j can distinguish the extension of E in which p_i takes a step first, then p_j, from the extension in which p_j takes its step before p_i. This implies that shared AND, OR and XOR registers cannot solve consensus, even for two processes. Consequently, their consensus number is 1.

Shared OR/XOR and AND/XOR registers

The algorithm presented in Figure 3 uses *or*() and *xor*() operations, but not the *and*() operation. It can easily be modified to use *and*() operations instead of *or*() operations. Lemma 12 trivially applies if the objects only offer *or*() and *xor*() (or *and*() and *xor*()) operations, and thus shared OR/XOR and AND/XOR registers have the same consensus number as shared AND/OR/XOR registers.

6 Conclusion

Bitwise operations are a common tool in sequential computing but, until now, they had not been considered in the distributed context. This paper studied their synchronization power by presenting the following contributions.

- The concept of shared AND/OR and AND/OR/XOR registers. A shared AND/OR register contains an array of x bits, to which it allows applying atomically bitwise AND and OR operations. A shared AND/OR/XOR register additionally offers the XOR operation.
- A wait-free algorithm that solves consensus for $\lfloor \frac{x+1}{2} \rfloor$ processes using x-bits shared AND/OR registers. This algorithm constitutes a lower bound on the consensus number of shared AND/OR registers.

- A modification of the previous algorithm that solves consensus for x processes using x-bits shared AND/OR/XOR registers. This algorithm constitutes a lower bound on the consensus number of shared AND/OR/XOR registers.
- A proof that x-bits shared AND/OR registers cannot solve consensus for more than $\lfloor \frac{x+1}{2} \rfloor$ processes. This constitutes an upper bound on the consensus number of shared AND/OR registers, and thus proves that the previous bound is tight.
- A proof that x-bits shared AND/OR/XOR registers cannot solve consensus for more than x processes. This constitutes an upper bound on the consensus number of shared AND/OR/XOR registers, and thus proves that the previous bound is tight.

These results show that shared AND/OR registers have consensus number $\lfloor \frac{x+1}{2} \rfloor$ and that shared AND/OR/XOR registers have consensus number x. Because bitwise operations are available in most modern processors, they can constitute a valuable tool for synchronization in distributed systems.

Acknowledgements. The author would like to thank Dmitry Kozlov for his help in the proof of Lemma 11.

References

1. 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 (PODC'06)*, pages 218–227, 2006. doi:10.1145/1146381.1146415.
2. Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing (PODC'93)*, pages 159–170, 1993. doi:10.1145/164051.164071.
3. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC'93)*, pages 91–100, 1993. doi:10.1145/167088.167119.
4. Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993. doi:10.1006/inco.1993.1043.
5. Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.
6. Eli Gafni and Petr Kuznetsov. On set consensus numbers. *Distributed Computing*, 24(3-4):149–163, 2011. doi:10.1007/s00446-011-0142-8.
7. Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. The synchronization power of coalesced memory accesses. *IEEE Transactions on Parallel and Distributed Systems*, 21(7):939–953, 2010. doi:10.1109/TPDS.2009.135.
8. Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 276–290, 1988. doi:10.1145/62546.62593.
9. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. doi:10.1145/114005.102808.
10. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.
11. Damien Imbs and Michel Raynal. A note on atomicity: Boosting test&set to solve consensus. *Information Processing Letters*, 109(12):589–591, 2009. doi:10.1016/j.ipl.2009.02.004.

12 Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4):592–614, 1997. doi:10.1145/263867.263888.

13 Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, 4:163–183, 1987.

14 Serge A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, pages 159–175, 1989. doi:10.1145/72981.72992.

15 Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000. doi:10.1137/S0097539796307698.

Wait-Free Concurrent Graph Objects with Dynamic Traversals

Nikolaos D. Kallimanis[1] and Eleni Kanellou[2]

1 FORTH-ICS, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece
nkallima@ics.forth.gr

2 FORTH-ICS, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece, and University of Rennes 1, Rennes, France
kanellou@ics.forth.gr

Abstract

Graphs are versatile data structures that allow the implementation of a variety of applications, such as computer-aided design and manufacturing, video gaming, or scientific simulations. However, although data structures such as queues, stacks, and trees have been widely studied and implemented in the concurrent context, multi-process applications that rely on graphs still largely use a sequential implementation where accesses are synchronized through the use of global locks or partitioning, thus imposing serious performance bottlenecks. In this paper we introduce an innovative concurrent graph model that provides addition and removal of any edge of the graph, as well as atomic traversals of a part (or the entirety) of the graph. We further present `Dense`, a concurrent graph implementation that aims at mitigating the two aforementioned implementation drawbacks. `Dense` achieves wait-freedom by relying on helping and provides the inbuilt capability of performing a partial snapshot on a dynamically determined subset of the graph.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases graph, shared memory, concurrent data structure, snapshot

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.27

1 Introduction

The irrevocable paradigm shift towards multi-core hardware has been accompanied by developments in concurrent data structure design. Data structures that are at the heart of many applications built for the sequential setting have been ported into the concurrent domain. Such data structures include trees (e.g. [2, 6, 10, 12]), stacks (e.g. [5, 14, 17, 23]) or queues (e.g. [16, 23, 25, 34]).

However, numerous are also the applications that rely on data structures with a more complex or irregular morphology. An example of such a data structure is the *graph*. A graph consists of a set V of *vertices* and a set E of *edges*, which are pairs of type (x, y), where x and y are elements of V. Each edge may additionally be associated with a value w, referred to as the *weight* of the edge, out of some set W. Vertices can be used to represent many types of entities, from simple or complex data structures to tasks, functions or processes, while the edges can flexibly express several types of relations. Graphs are essential building blocks for applications in robotics (e.g. [9]), machine learning (e.g. [36]), automated design of digital circuits (e.g. [24]), task scheduling in operating systems (e.g. [27]), garbage collection (e.g. [39]), and video-game design (e.g. [7]) to name a few.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 27; pp. 27:1–27:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Thus, in the multi-core era, applications that rely on graphs are also important in the concurrent context, where they can be used either in message-passing or in shared memory settings. Notable shared memory examples include Computer-Aided Manufacturing (CAM) applications, scientific simulation tools, or video games, where a virtual world is represented as a graph. Vertices of the graph model objects in the virtual world, while edges represent the relationships and interactions of those objects [3]. As these relationships and interactions are unlikely to be static, this implies that processes executing the application need to update the structure and connectivity of the graph frequently. However, concurrent applications that rely on graphs still mostly use sequential implementations and synchronize the access of multiple processes to them with the help of a global lock or by partitioning the vertices into disjoint sets that can then individually be accessed by different processes [37].

If a global lock is used, accesses become expensive and the lock poses a global bottleneck for overall performance. Furthermore, even if fine-grained locking is employed on subsets of the graph, the liveness of the application is restricted, as it becomes blocking by the reliance on locks. By relying on graph partitioning instead, the interaction of processes for the purpose of synchronizing accesses to the graph can be minimized, as different processes access distinct partitions. Such an implementation can offer a higher degree of parallelism to applications that are suitable for exploiting it. However, such an application then either has to rely on a static, predefined partitioning, or it has to perform re-partitioning in order to accommodate dynamicity, in which case synchronization among all processes may be necessary and performance is likely to suffer.

These issues may be aggravated when the operation to be performed affects the entire graph or a subset of it, as is the case with iterations, searches, or partial traversals. In those cases it is important to provide a consistent view of the graph or some subgraph for the operation, but without blocking or impeding concurrent updates on it. This possibility is offered by atomic snapshots [1]. A snapshot is a concurrent object that consists of a collection of components and which provides two operations: `Update`, which modifies the value of one of the components, and `Scan`, which returns a consistent view of the values of all components. To implement a graph, a one-to-one correspondence is established between graph edges and snapshot components. Modifications on the graph structure are performed by using the `Update` operation of the snapshot, while reads are done by using `Scan`.

When only a subset of the graph has to be accessed, the overhead that a snapshot incurs can be avoided by using partial snapshot implementations, such as [4, 22]. There, `Scan` takes as input argument the subset of components on which the snapshot is to be performed. However, this precludes applications, in which the subgraph that should be accessed is not known a priori, e.g. when there is the need to perform a random traversal on some part of the graph, where the next edge to be accessed is determined dynamically.

In this paper we are concerned with the problem of providing such a dynamic traversal for graphs. A generalized solution consists in using *transactional memory* (TM) [20, 35]. This paradigm offers the *transaction* abstraction. A transaction encapsulates one or several read and update operations and attempts to apply them atomically. If this is possible, the transaction *commits* making the effects of the operations visible. Otherwise, it *aborts* discarding any change. We want to avoid the commit/abort semantics inherent in this paradigm, the performance overheads that this paradigm can entail, as well as well-known progress limitations [8]. For this reason, we define a novel concurrent graph object. It supports the addition, removal, and modification of edges of the graph, as well as the atomic walk of a dynamically-determined subset of the graph edges, such that a consistent view of it may be obtained. The model we propose for the dynamic traversals is reminiscent of TM:

Dynamic traversals resemble read-only transactions, while modifications of the edges of the graph behave like mini-transactions that encapsulate a single update on a single transactional object. Contrary to common transactional semantics, however, our model does not need, and so does not include the possibility to abort.

We further present `Dense`, a adjacency matrix concurrent graph implementation based on the proposed model. `Dense` provides the linearizable operation `Update`, which is used to add or remove edges of the graph or to modify existing edge weights. The *step complexity* (i.e., the total number of accesses in the shared base objects) is $O(k)$, where k is the number of active processes (i.e. `Dense` is an adaptive algorithm). Dynamic traversals are implemented as a composite operation, reminiscent of read-only transactions. For this purpose, `Dense` provides `DynamicTraverse`, another linearizable operation that is used to initiate a dynamic traversal, and a matching `EndTraverse` operation which terminates it. An auxiliary routine, `Read`, is used for obtaining edge weights. Instances of `Read` that are enclosed by an instance of `DynamicTraverse` and a matching instance of `EndTraverse`, return a consistent view for the weights of the edges they read. As such, dynamic traversals appear to be atomic and can be considered as a virtual operation that is linearizable.

`Dense` operations are *wait-free*, i.e. an operation by a process that does not fail terminates in a finite number of steps in any execution. Wait-freedom is achieved by employing light-weight *helping* using a mechanism reminiscent of the one presented in [19]. Instances of operations that are concurrent with instances of `Update` help them carry out edge modifications. Operations are aware of concurrent active dynamic traversals and ensure that those dynamic traversals can return a consistent view by storing old edge versions for them (in the worst case, `Dense` keeps n different versions, one for each process, on a given edge of the graph). The step complexity of `DynamicTraverse` and `Read` is $O(k)$ and $O(1)$, respectively. `Dense` uses m `LL/SC` base objects (one for each edge), one atomic *Add* base object of n bits, and an additional `LL/SC` object. `Dense` is of theoretical interest since the size of `LL/SC` objects is big. We further present `S-Dense`, a practical version of `Dense`, with step complexity $O(k^2)$, which uses small base objects.

The rest of the paper is organized as follows. Section 2 summarizes relevant literature. Section 3 provides our view of the underlying system. Section 4 gives an overview of the defining characteristics of the proposed implementation, while Section 5 provides a detailed description of it and sketches out the proof of the correctness argument. Finally, Section 6 contains a discussion of the presented results and of prospects for future work.

2 Related Work

A great body of work on the concurrent implementation of graph algorithms tackles common graph-related issues (e.g. [11, 29, 33]) and focuses either on parallelizing existing sequential algorithms or on providing concurrency through the use of locks on well-known sequential algorithms. Then, liveness guarantees are rather relaxed, as most of these implementations are blocking. In contrast, we are interested in the graph as a general-purpose, concurrent data structure and are especially concerned with providing wait-freedom and linearizability.

Work on concurrent data structures has been devoted to commonly-used ones, such as queues, stacks, or trees, with the focus on providing interesting progress properties – initially simply by avoiding locks (e.g. [28, 34]), and recently a step further, by proposing wait-free implementations. Notably, [25] uses helping via an announce array in order to make a wait-free version out of the `CAS`-based lock-free queue of [28]. Together with a “fast path, slow path” methodology [38], previously used for the implementation of a wait-free linked list out of well-known lock-free design [15], this method is proposed as a generalized methodology of designing wait-free concurrent data structures, given a lock-free implementation [26]. Our method is “stand-alone”, providing wait-freedom without requiring a lock-free design as base.

Recently, techniques that provide *iterators* of concurrent data structures have been proposed. An iterator parses a data structure in order to obtain a consistent view. In [31], a methodology is proposed for enhancing lock-free or wait-free set-based data structures with a `CAS`-based implementation of a wait-free iterator. It entails reporting data structure updates to any active snapshot, so that they can be taken into account, depending on the order of linearization. In [32], update and read operations on a trie are aware of an ongoing iterator, and copy – and thus, effectively rebuild – the parts of the trie that they access, leaving intact the albeit obsolete version that the iterator is parsing. The complexity is divided among updates and reads, while the snapshot occurs in constant time. In [30] a theoretical framework for defining the consistency of iterators is proposed and used in a case study that equips the well-known lock-free concurrent queue of [28] with a wait-free iterator that is linearizable according to the provided framework.

We, however, are interested in a partial view, which, furthermore, is dynamically defined. Thus, we want to avoid the overhead that is induced by iterating over the entire data structure. Arguably, the implementation in [32] does not induce it, having a constant-time snapshot. However, to achieve that, it must employ either double compare, single swap (`DCSS`) primitives, or a custom-made, `CAS`-like software primitive,unlike our method, which simply relies on `LL/SC`. Moreover, those works take advantage of the structural regularity of the underlying data structure. In contrast, a graph usually has irregular characteristics. Our work is more akin to partial snapshots, such as [4, 22], as we use an adjacency matrix to represent the graph. However, partial snapshots are more restrictive than our model as they require a priori knowledge of the component subset to be scanned.

The required dynamicity can be provided by using transactional memory to access a graph. Indeed the dynamic traversal provided by our model resembles a read-only transaction. However, efficient TM algorithms commonly rely on locks, while even obstruction-free or non-blocking ones commonly burden reads and updates with the processing overhead necessary for conflict-detection and resolution (cf. with [13] for a survey on TM algorithmic techniques). We wish to avoid these issues, as well as the commit/abort semantics inherent in TM, but unusual for data structures. The recent impossibility result in [8] further implies that, even if commit/abort semantics are included in our model, the TM progress property equivalent to wait-freedom cannot be achieved.

3 Model

System model. We assume an asynchronous, shared memory system of n processes, which communicate by accessing *base objects*. A base object O has a state and provides a set of *primitives*, used by processes in order to access, i.e. read and/or modify, the state of O. We use the following base objects: A read/write object O has a state that takes values out of some set S. It provides the primitives $read(O)$, which returns the state of O, and $write(O, v)$, $v \in S$, which sets the state of O to v. An *Add* object O has a state that takes values out of some set of integers S. It provides the primitives $read(O)$, which returns the state of O, and $add(O, v)$, $v \in S$, which adds the value v to the state of O. An `LL/SC` object O has a state that takes values out of some set S. It provides the primitives $\texttt{LL}(O)$ and $\texttt{SC}(O, v)$, $v \in S$. $\texttt{LL}(O)$ returns the current state of O. $\texttt{SC}(O, v)$, executed by a process p_u, $u \in \{1, 2, \ldots, n\}$, must follow an execution of $\texttt{LL}(O)$ also by p_u. It changes the state of O to v if O has not changed since the last execution of LL(O) by p_u. The concurrent implementation of a data structure also has a *state*, stored in shared base objects. It provides algorithms for the operations provided by the data structure, which processes may use in order to access or modify its state. A process executes an operation by issuing an *invocation* for it and an operation terminates by returning a *response* to the process.

Executions. At any point in time, the system is characterized by a *configuration* C, which is a vector that contains the state of each process in the system and the state of each base object. We denote by C_0 the initial configuration of the system. A *step* ϕ is either the execution of a primitive by some process, or the issuing of an operation invocation by some process, or the response of some operation to some process.

A (possibly infinite) sequence $C_0, \phi_1, C_1, \ldots, C_{i-1}, \phi_i, C_i, \ldots$, of alternating configurations (C_k) and steps (ϕ_k), starting from the initial configuration C_0, where for each $k \geq 0$, C_{k+1} results from applying step ϕ_{k+1} to configuration C_k, is referred to as an *execution*. A subsequence of an execution α in the form $C_i, \phi_{i+1}, C_{i+1}, \ldots, C_j, \phi_{j+1}, C_{j+1}$, of alternating configurations and steps, starting from some configuration C_k, $k > 0$, is referred to as an *execution interval* of α.

If some configuration C occurs before some configuration C', $C \neq C'$, in an execution α, then we say that C *precedes* C' in α and denote it as $C < C'$. Conversely, we say that C' *follows* C in α. Precedence among a step ϕ_i and a step ϕ_j, or precedence among a step ϕ_i and a configuration C_j is defined in a similar fashion and denoted by the same operation $<$.

Let α_1 and α_2 be two execution intervals of some execution α. If the last configuration of α_1 precedes or is the same with the first configuration in α_2, then we say that α_1 precedes α_2 and denote it $\alpha_1 < \alpha_2$. In that case we also say that α_2 follows α_1. If neither $\alpha_1 < \alpha_2$ nor $\alpha_2 < \alpha_1$ are the case, then we say that α_1 and α_2 *overlap*.

Given the instance of some operation *op* for which the invocation and response steps are included in α, we define α_{op}, the execution interval of *op*, as that subsequence of α which starts with the configuration in which *op* is invoked and ends with the configuration that results from the response of *op*. If there are no two operation instances op_1, op_2 in α for which the execution intervals overlap, then we say that α is a *sequential* execution, or that operations in α are executed *sequentially*.

Concurrent graph. A *graph* $G = \langle V, E \rangle$ is composed of V, a (finite) set of elements referred to as *vertices*, and E, a set of pairs of vertices, referred to as the *edges* between them. Each edge $e_{i,j} \in E$ has a weight w_{ij}, that takes values out of some set W. A graph supports several abstract operations, well-known in literature, such as operations for adding vertices or edges, deleting vertices or edges, modifying attributes of vertices or edges, returning specific subsets of the graph vertices or edges, etc. A *concurrent graph* is a graph that can be accessed concurrently, through those types of operations, by n processes.

We propose the *dynamic traversal* (henceforth referred to as *d-traversal* for brevity) as a concurrent graph operation exhibiting the following characteristics: (i) starts from a vertex v of the graph, (ii) visits a sequence of vertices that is not necessarily known at the point that the traversal initiates, (iii) the sequence of visits may be decided while the visiting is taking place; (iv) the dynamic traversal returns a *consistent view* of the weights of all the edges that it has traversed, i.e., all the returned values have co-existed at some point in time.

We further propose a concurrent graph implementation. The graph is represented as an $m \times m$ adjacency matrix, for some positive integer m, and it allows the addition, and removal of edges, the modification of edge weights by providing an `Update` operation. `Update`(i, j, w), where i, j are indices of vertices in V and where w is in $W \cup \{\bot\}$, modifies the graph as follows: Assume that $e_{i,j} \in E$. If $w = \bot$, then the edge is removed. Otherwise, its weight is changed to w. If $e_{i,j} \notin E$, then it is inserted in E with weight w. The implementation supports the *d-traversal* as a composite operation, consisting of the following ones:

- `DynamicTraverse`, which is used to mark the beginning of a *d-traversal* of the graph.
- `EndTraverse`, which is used to mark the end of a *d-traversal* of the graph.
- `Read`(i, j), where i, j are indices of vertices in V. It returns a weight for edge $e_{ij,}$, if $e_{i,j} \in E$, and $\bot$ if $e_{i,j} \notin E$.

`Read` is only used in *d-traversals*, as part of a sequence of `Read` operations. A *d-traversal* by process p_u consists in an instance *bt* of a `DynamicTraverse` operation, followed by a sequence of instances of `Read`, followed in turn by an instance *et* of an `EndTraverse` operation. No other operation is invoked between *bt* and *et*. The execution interval of the *d-traversal* starts in the configuration in which p_u invokes *bt* and ends in the configuration resulting from the response of *et*. For correctness argumentation, we consider *d-traversal* as a virtual operation that is invoked at the point that `DynamicTraverse` is invoked and responds at the matching `EndTraverse` (if any).

Correctness criteria. We consider *linearizability* [21] as correctness criterion for the graph operations. An execution α is *linearizable* if it is possible to assign a *linearization point* inside the execution interval of each operation in α (`Update` and *d-traversal* operations), so that the result of these operations is the same as it would be, if they had been performed sequentially in the order dictated by their linearization points. A *d-traversal* operation is considered linearizable if it is possible to assign a linearization point in its execution interval so the result of the `Read` operations enclosed in it, is the same as it would be, if all the `Read` operations had been performed at the linearization point in the sequential execution (the order of `Read` is imposed by the invocation steps). Roughly speaking, we consider that the entire sequence of `Read` operations enclosed in a dynamic traversal have a linearization point inside the execution interval of the *d-traversal*, such that the `Read` return the weights that the traversed edges had in the configuration in which the linearization point is placed.

Progress criteria. In this work we consider that processes that participate in an execution α may suffer from *crash failures*, i.e. we consider that a process may unexpectedly stop taking steps in α after some configuration. In this context, we provide a graph implementation with operations that satisfy *wait-freedom* [18]. A data structure implementation is wait-free if in any execution, each process finishes the execution of every operation it initiates within a finite number of steps independently of the speed or the failure of other processes.

4 Main Ideas

Our implementation provides linearizable, wait-free operations and linearizable *d-traversals* by using light-weight helping. To achieve it, each `Update` or `DynamicTraverse` operation is first *announced* by a process, subsequently *agreed* by all processes, and then it can be *applied* by some process – not necessarily the one that invoked it – and finally, terminate and return a response. Processes provide very light-weight assistance to each other when applying operations. In the case of `DynamicTraverse` operations, helping consists in storing an integer number for it, while in the case of `Update`, helping consists in applying the update on the edge. In order to coordinate how announced operations are executed among multiple processes, the processes collaborate to alternate between two types of phases, namely AGREE and APPLY. An announce array of dimension n, an n-bit bitvector *Add* object, and two sets of n bits each, namely *ann* and *done*, are used for this coordination.

To announce an operation, a process p_u, $1 \leq u \leq n$, starts by writing the operation type and arguments (the *operation information* or op-info for brevity) in the u-th element of the announce array. Notice that p_u is the only process that may write to this element while all processes in the system can read it. Subsequently, p_u flips the value of the u-th bit of the bitvector. Other processes can now also help the operation of p_u.

The AGREE phase is used by processes in order detect which op-info in the announce array corresponds to a pending operation: p_u has a pending operation if the u-th bit of the bitvector is not equal to $done[u]$. In this phase, processes essentially "agree" on a set of operations that they will attempt to apply on the graph in the following APPLY phase. Then, the APPLY phase that follows is used by processes for attempting to apply those pending operations. As a result, operations are applied to the graph in batches. When an announced operation is carried out by some process, we say that it is *applied*. Otherwise, it is *pending*. An applied operation can return a response to the process that invoked it. The status of an operation, i.e. whether it has been already applied or not, is reflected in the values of $ann[u]$ and $done[u]$: An invariant in our implementation is that when $ann[u] = done[u]$, the latest agreed operation by p_u has been applied, while when $ann[u] \neq done[u]$, it is pending. A process which completes the actions associated with a phase, attempts to flip it.

A shared integer *seq* acts as global version counter and is also used to make the wait-free partial traversals possible. Each time the phase changes from AGREE to APPLY, *seq* is incremented. Apart from their weight, edges of the graph also have a version number as a further attribute. In each configuration, the version of an edge is the value of *seq* at the configuration in which the edge was last updated. Each *d-traversal* that is initiated, is also assigned – either by the process that executes it or by some other, helping process – the value that *seq* has at the configuration in which its DynamicTraverse is applied. This value is referred to as the *d-traversal read value* and is visible to all processes.

Before a process p_u applies an Update on edge $e_{i,j}$, it also detects whether a *d-traversal* by some other process p_l, $1 \leq u \leq n$, is active. If the *read version* of p_l's *d-traversal* is lower than the current version of $e_{i,j}$, p_u stores the weight and version of $e_{i,j}$ for p_l – as *previous weight* and *previous version* for p_l – and updates them only afterwards. For this, each edge contains a vector *prev* of dimension n, where element l corresponds to p_l and contains the previous weight and previous version that p_l may need during its *d-traversal*.

When p_l uses Read to collect the weight-version pair of an edge that it accesses during a *d-traversal*, it compares the edge's version with the *read version* of the *d-traversal*. If it is greater, p_l collects the previous weight and previous version of the edge. Otherwise it uses the current weight and value. This way, all the Read that are enclosed between the DynamicTraverse and the matching EndTraverse operation of p_l's *d-traversal*, return mutually consistent values and the *d-traversal* forms a consistent view.

5 Dense, a Concurrent Graph Implementation

In the following, Dense, an algorithm for our proposed concurrent graph implementation, is presented. Dense is so named because it is mostly suitable for dense graphs, i.e. graphs with high connectivity, in which case the allocated adjacency matrix is sufficiently exploited.

Listing 1 shows the data structures used by Dense (initial values are indicated on lines 20–23). Operation information is stored in a structure of type *AnnStruct*. This structure

```
type OpType = {DynamicTraverse, Update, Noop}; //operation types

struct AnnStruct // the data type of the announce array elements
  OpType op; // the  announced operation
  int i, j; // if OpType=Update, $e_{i,j}$ has to be updated
  int value; // weight to be assigned to $e_{i,j}$ if OpType=Update
};

struct StateStruct // data type for storing the graph's state
  int seq; // the sequence number, used as a version counter
  boolean phase; // current phase of execution, Announce or Apply
  int ann[1..n]; // used as $n$-bit vector
  int done[1..n]; // used as $n$-bit vector
  int rvals[1..n]; // read value for each process
};

struct EdgeStruct { // the data type of a graph edge
      ⟨ weightval, int ⟩ prev[1..n]; // one element per process
      int seq; // current version of the edge
      weightval w; // current weight of the edge
};

shared int BitVector; // used as $n$-bit vector
shared AnnStruct Announce[1..n] = {$\langle\langle \mathrm{Noop}, 0, 0, 0\rangle, \ldots, \langle \mathrm{Noop}, 0, 0, 0\rangle\rangle$};
shared StateStruct ST = $\langle 0, \mathrm{AGREE}, \langle 0, \ldots, 0\rangle, \langle 0, \ldots, 0\rangle, \langle 0, \ldots, 0\rangle\rangle$;
                                      // operations status and phase indicator
shared EdgeStruct Edges[1..m][1..m] = {$\langle\langle\langle 0,0\rangle, 0, 0\rangle, \ldots, \langle\langle 0,0\rangle, 0, 0\rangle$};
                                    // adjacency matrix representing the graph
private int $toggle_u$ = $2^u$; // a variable per process, $u \in \{1, \ldots, n\}$
```

Listing 1 Dense: Data structures for a concurrent graph object suitable for dense graphs.

consists of four fields, namely: (i) *op*, of type *OpType*, which represents operations provided by Dense (i.e., DynamicTraverse, Update, and the void operation Noop); (ii) i and j which identify the edge on which an Update operation is to be applied (if op = Update); and (iii) *value*, an integer representing the value that an Update operation has to apply to the weight of the edge specified by fields i and j (if op = Update).

The status of operations on the graph is indicated by ST, an LL/SC object of type *StateStruct* consisting of: (i) *seq*, an integer which serves as global version counter. It is incremented each time a process successfully switches the execution phase from AGREE to APPLY; (ii) *phase*, a boolean variable which indicates whether the execution of Dense is in an AGREE or an APPLY phase at any given moment; (iii) $ann[1..n]$, an array implemented as n-bit integer, where $ann[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and whose value is toggled each time an operation by p_u is agreed; (iv) $done[1..n]$, an array implemented as n-bit integer, where $done[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and whose value is set equal to $ann[u]$ each time an operation by p_u is applied to the graph; and (v) $rvals[1..n]$, an array of n elements, where t $rvals[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and which stores the value of *seq* that p_u uses as *read version*, in case it is performing a *d-traversal*.

We represent the graph G with *Edges*, an adjacency matrix, i.e. a two-dimensional array, where each element (i, j) of the array represents edge between vertices i and j, $i, j \leq m$. Graph edges, i.e. adjacency matrix elements, are LL/SC objects of type *EdgeStruct*. This type is a record of three fields: (i) *prev*, an array of n elements (one for each process),

```
void Update(int i, int j, int value) { //for process p_u, u ∈ {1,...,n}
   BTU(Update, int i, int j, int value)
}

void DynamicTraverse() { //for process p_u, u ∈ {1,...,n}
   BTU(DynamicTraverse, ⊥, ⊥, ⊥);
}

void EndTraverse() { //for process p_u, u ∈ {1,...,n}
   ;
}

int Read(int i, int j) { //for process p_u, u ∈ {1,...,n}
   EdgeStruct edge;
   int val, int seq, int rval;
   edge = Edges[i][j];
   rval = ST.rvals[u];

   if (edge.seq > rval) {
      ⟨val, seq⟩ = edge.prev[u];
   }
   else val = edge.w;
   return val;
}
```

Listing 2 `Dense`: Operations `Update`, `DynamicTraverse`, `EndTraverse`, and `Read`.

where each element is a pair $< w, seq >$ of integers. Whenever an update operation modifies the weight of an edge, it stores the current weight and version in $prev[u]$ if process p_u is performing a *d-traversal* on the graph using as read value, stored in $ST.rvals[u]$, a value that is larger than the current version of the edge; (ii) seq, an integer which stores the current version of the edge; (iii) w, of type *weightval*, which stores the current weight of the edge - recall that if this value is $\perp$, the corresponding edge does not exist.

Recall that `Dense` implements the helping mechanism, where any process p_u that invokes an operation also attempts to apply pending operations by other processes. Operation information is stored by processes in $Announce[1..n]$, an announce array of n elements, where each element $Announce[u]$, $u \in \{1, 2, \ldots, n\}$, is of type *AnnStruct* and can be written to only by process p_u, but can be read by all processes. The announcing of an operation is complemented by the use of $BitVector$, shared vector of n bits (represented as a n-bit integer) where bit u corresponds to process p_u. In order to indicate a pending operations, each time p_u announces operation information in $Announce[u]$, it flips the u-th bit of $BitVector$. It does so with the aid of a local, persistent variable, $toggle_u$, with initial value 2^u. After p_u announces an operation, it inverts the value of $toggle_u$.

Pseudocode Description. Pseudocode for the operations of the graph that are described in Section 3 is presented in Listing 2. Operations `Update` and `DynamicTraverse` require that the processes that execute them, assist each other. In order to do this, they both invoke auxiliary routine `BTU` (these initials stand for "*B*egin a *T*raversal or *U*pdate"). `BTU` implements the phase alternation and is further detailed below. We say that an execution of `Dense` is in `AGREE` or `APPLY` phase during those execution intervals in which $ST.phase =$ `AGREE`, or $ST.phase =$ `APPLY`, respectively. Notice that `Read` is independent of the phases. Instances of

Read are only invoked by a process following the execution of a DynamicTraverse operation by the same process. They rely on Update operations to store possibly useful old edge versions for them in the *prev* arrays of each modified edge.

The DynamicTraverse operation that initiates some *d-traversal* d, obtains as *read version* the current value v of $ST.seq$ (this happens when either the process that initiated d or some other process helps to apply this DynamicTraverse operation while executing line 74). An instance r of Read that is invoked by process p_u on edge $e_{i,j}$ and that is included in d, must check whether the version of $e_{i,j}$ is greater than v (line 38). If this is the case, then $e_{i,j}$ was updated after d started. However, in Dense, *d-traversals* must not be aware of concurrent edge updates and have to return values that the edge weights had before the *d-traversal* initiated. For this reason, r must return a previous weight of $e_{i,j}$, and finds this in $e_{i,j}.prev[u]$ (line 39). If the version of $e_{i,j}$ is less than v, then r returns $e_{i,j}$'s current weight (line 41). Notice that although the instances of Read that are included in a *d-traversal* are not aware of concurrent Update instances (i.e. instances whose execution intervals overlap with that of the *d-traversal*), those Update instances become aware of *d-traversals* and store the necessary old edge weights for them when they modify edges the graph.

Listing 2 presents BTU, which is at the heart of the Dense implementation. It is invoked by Update specifying as arguments the operation type, integers i and j, which identify the edge to be modified, and integer *value*, which specifies the weight to be written to this edge. When BTU is invoked by DynamicTraverse, then only the operation type is specified as argument, while the remaining three are $\perp$, as they are not required for the *d-traversal.*

An instance of BTU by p_u first writes the operation information into element u of the announce array (line 48) and then sets the value of the u-th bit of $BitVector$ (line 49), using the current value of local persistent variable $toggle_u$ bit. It then flips $toggle_u$ (line 50) in order to prepare its value for the next execution of an operation by p_u. The algorithm implements this practice in order to provide a previously mentioned invariant: by comparing $ST.ann[u]$ and $ST.done[u]$, a process is able to detect whether the latest agreed operation by p_u has already been applied or not. Notice that the contents of $BitVector$ are copied into $ST.ann$ by each process that successfully executes an AGREE phase of Dense (lines 53, 56, 80), while they are copied into $ST.done$ by a process that successfully executes an APPLY phase of Dense (lines 53, 77, 80). Therefore, each operation by p_u must correspond to a different $BitVector[u]$ value than the previous one.

BTU carries out any light-weight helping in addition to the execution of the operation that invoked it. To do this, it iterates via a for loop (lines 51-81). An iteration of this for loop consists in locally copying ST (line 52), and then attempting to perform the actions that are required by the phase indicated in $ST.phase.$ Once these actions have been performed, BTU attempts to change the phase by executing the SC of line 80. If this SC is successful, we say that BTU (or, abusing terminology, the process or the operation that invoked it) successfully executed the phase. The execution of this primitive may fail if some instance of BTU, executed by a process other than p_u, has already performed the current phase and advanced the execution to the next phase. When executing the for loop (lines 51-81), BTU proceeds as follows, depending on the phase it performs:

- AGREE phase (lines 55-58). This phase updates the status record ST with the newly announced operations, so that all processes can agree on them. So, BTU first records this status locally on *st*, before using an SC instruction in order to attempt to update it globally on ST. In order to set *st*, BTU collects information from the $BitVector$ regarding newly announced and therefore possibly pending operations. It does so by copying the contents of $BitVector$ into *st.ann* (line 56). Notice that for a process $p_l, 1 \leq l \leq n$

```
44 void BTU(OpType op, int i, int j, int value) { //for p_u, u ∈ {1,...,n}
45   StateStruct st;
46   int lbv, opi, opj;
47   EdgeStruct e;

48   Announce[u] = ⟨op, i, j, value⟩;
49   Add(BitVector, toggle_u);
50   toggle_u = - toggle_u;

51   for (i=0; i < 4; i++) {
52     st = LL(ST);
53     lbv = BitVector;

54     if (lbv[u] == st.done[u]) break;

55     if (st.phase == AGREE) {  // AGREE Phase
56       st.ann[1..n] = lbv[1..n];
57       st.seq = st.seq + 1;
58       st.phase = APPLY;
59     } else {  // APPLY Phase
60       for (r = 1; r ≤ n; r++) {
             // at most k shared memory accesses, k=active processes
61         if (st.ann[r] ≠ st.done[r]) {
62           if (Announce[r].op == Update) {
63             opi = Announce[r].i;
64             opj = Announce[r].j;
65             e = LL(Edges[opi][opj]);
66             if (e.seq < st.seq) {
67               for (k = 1; k ≤ n; k++) {
68                 if (e.seq < st.rvals[k]) e.prev[k] = ⟨e.w, e.seq⟩;
69               }
70               e.w = Announce[r].value;
71               e.seq = st.seq;
72               SC(Edges[opi][opj], e);
73             } // if (e.seq < st.seq)
74           } else st.rvals[r] = st.seq;
75         } // if (st.ann[r] ≠ st.done[r])
76       } // for (r = 1; r ≤ n; r++)
77       st.done[1..n] = lbv[1..n];
78       st.phase = AGREE;
79     }
80     SC(ST, st);
81   }
82 }
```

Listing 3 Dense: BTU auxiliary routine.

that has a newly announced operation, the invariant $st.ann[u] \neq st.done[u]$ must hold. Therefore, a successful assignment of st to ST (through the execution of the SC of line 80) creates the inequality between $ST.ann[u]$ and $ST.done[u]$ and makes all processes "agree" that p_u has a newly announced operation which has not been applied yet. Once the information regarding pending operation for each process has been copied into st, BTU increments seq, the global version counter in st (line 57) and changes the $phase$ field of st from AGREE to APPLY.

- APPLY phase (lines 59–78). This phase applies any pending agreed Update operation on the edges of the graph, and assigns *read version* to any pending agreed DynamicTraverse operation. For this, BTU uses st again, and for each process p_u (line 60) it checks whether such a pending operation exists (line 61), in which case it holds that $st.ann[u] \neq st.done[u]$. Consider the case of a pending Update operation by p_u on edge $e_{i,j}$. Since multiple processes may be executing an operation on $e_{i,j}$, these modifications must be synchronized in order to safeguard correctness. For this reason, $e_{i,j}$ is copied locally into e using LL (line 65). If the current version number of $e_{i,j}$, $e.seq$ is greater than $st.seq$ then the specific Update operation has already been applied, namely by some process other than p_u, that has also changed the state. However, if this is not the case, the modification of $e_{i,j}$ is carried out. Before setting the new value for the weight (line 70) and version (line 71) of $e_{i,j}$, a comparison of the current version of $e_{i,j}$ and all *read versions* stored in $st.rvals$ is performed (lines 67–68). If the current version of $e_{i,j}$ is less than the *read version* for some process p_r, $1leqr \leq n$, then the condition $e.seq<st.rvals[r]$ is true. This means that a concurrent *d-traversal* by process p_r might be in progress. In order to guarantee that an eventual such *d-traversal* can read mutually consistent values, the current values of $e_{i,j}$'s weight and version are stored in $e.prev[r]$. There, instances of Read on $e_{i,j}$ that are included in a *d-traversal*, can later find it if necessary. BTU then attempts to finalize the update of $e_{i,j}$ by using SC to copy e into $e_{i,j}$ (line 72). Whether the SC on the edge is successful or not, the operation is considered applied.
  If p_u's pending operation is a DynamicTraverse, the *read version* must simply be set. This is first recorded in $st.rvals[u]$ (line 74) and is eventually stored in $ST.rvals[u]$ (line 80) by the process that successfully executes the phase. Recall that it is used by a concurrent Update operation in order to judge whether to discard the current value of the edge that it is updating or whether to keep it for the ongoing *d-traversal* of p_u. If the assignment of line 74 followed by a successful SC on ST is executed more than once for a given DynamicTraverse instance or for the *d-traversal* that it initiated, then the consistency of the Read instances of the *d-traversal* could be compromised. An eventual bad scenario would happen if Read instances that are invoked before the second execution of those lines and Read instances that are invoked after the second execution would use a different *read version* when reading edges.
  Thus, at the end of an APPLY phase, the *done* bits in st are set equal to the corresponding *ann* bits (line 77). Then, BTU attempts to change the phase from APPLY back to AGREE (line 78) by switching the *phase* field of st, which is reflected on ST if the SC instruction of line 80 is successful.

Notice that an instance of BTU may be slow and end up performing the actions associated with a phase while the execution has already progressed to the next phase. Notice also that in the worst case, an instance of BTU has to perform four iterations of the for loop before the operation that invoked it is applied. Such a worst-case scenario is the following: Let I_{btu} be an instance of BTU that executes the first iteration of the for loop during an AGREE phase and let p_l be the process that successfully flips the phase to APPLY by executing the SC on ST of line 80. Consider however that the execution of line 49 by I_{btu} occurs after p_l executes the LL of line 52, which corresponds to the successful SC on ST. This means that in the following APPLY phase, the operation that invoked I_{btu} will not be executed. In the worst case, all other processes are slow and the process that invoked I_{btu} must perform the actions associated with the APPLY phase itself, during the second iteration of the for loop, as well as the actions required by the following AGREE phase, during the third iteration of its for loop. During this AGREE phase, the *Add* on $BitVector$ by I_{btu} is guaranteed to be observed by the process that performs the successful SC on ST and changes the phase to APPLY. Here again, in the worst case, all other processes are once more slower than the process which invoked I_{btu}, and thus it is I_{btu} that performs the actions associated with the APPLY phase, in its fourth iteration of the for loop. This time, however, the operation that invoked it is guaranteed to have been applied.

However, in the common case, the operation may be applied earlier, by some other, helping process. The condition that signals this is expressed on line 54 and is checked at each iteration of the for loop. It consists in verifying whether the toggle bit for p_l, the process executing BTU, in shared array $BitVector$ has the same value as the corresponding bit in the $ST.done$ array. If that is the case, the operation executed by BTU is considered applied and the iteration of the for loop terminates as well.

S-Dense. We provide S-Dense, a variation on Dense, implemented with smaller registers. S-Dense is meant to conform to current machine architectures restrictions. The main difference with Dense is the implementation of the $EdgeStruct$, which no longer contains a large register $prev$. Instead, a 3-dimensional array $prev$, external to the $EdgeStruct$ structure, is used. A process that updates some edge $e_{i,j}$ records locally the weight w_{old} and version seq_{old} that it read in $e_{i,j}$. After performing the SC on $e_{i,j}$ (as on line 72 of Dense), the process uses the LL/SC primitive to attempt to write w_{old} and seq_{old} into the $prev$ position they have to be recorded for some other traversing process. Notice that the status LL/SC object ST can also be implemented with smaller registers, by applying the technique that is used in the P-Sim algorithm that is presented in [14]. A detailed description of S-Dense will be presented in the full version of the paper.

5.1 Proof of Correctness Sketch

Due to space constraints, we present a sketch of the correctness argument for our algorithm. Consider an execution α of Dense. Denote by $SC_1^{ST}, SC_2^{ST}, \ldots$ the sequence of successful SC of line 80 on ST in α and by $LL_1^{ST}, LL_2^{ST}, \ldots$ the sequence of matching LL on ST. We first prove that the phases of Dense indeed oscillate between AGREE and APPLY.

▶ **Corollary 1.** *Any SC_k^{ST} such that $k \mod 2 = 1$ changes $ST.phase$ from AGREE to APPLY. Any SC_k^{ST} such that $k \mod 2 = 0$ changes $ST.phase$ from APPLY to AGREE.*

Let op be an instance of some operation, invoked by process p_u, $1 \leq u \leq n$. When line 49 is executed for op, we say that op is *announced*. Let op be an operation that is announced in some configuration C. Let p_l, $1 \leq l \leq n$ where possibly $l \neq u$, be a process which executes a successful SC on ST after C, such that the value written to $BitVector[u]$ at C is copied into $ST.toggles[u]$. If this occurs, we say that op has been *agreed*. Let op be a DynamicTraverse operation that is agreed in some configuration C'. Let p_l, where possibly $l \neq u$, be a process which executes a successful SC on ST after C', such that the value written to $BitVector[u]$ at C is now also copied into $ST.done[u]$. If this occurs, we say that op has been *applied*. We say that an agreed Update operation op, with edge $e_{i,j}$ as parameter, has been applied, if some process p_l successfully executes the SC of line 72 on $e_{i,j}$ with the parameter v of op. We assign the linearization points to operations in the configuration in which they are applied.

We prove that each operation is agreed in its execution interval and use this to prove that it is also applied exactly once during its execution interval, as well as the following lemmas:

▶ **Lemma 2.** *Let OP be any instance of either DynamicTraverse or Update. The linearization point of OP is included in its execution interval.*

Then, we prove that instances of `Read` enclosed in a *d-traversal*, read edge values that are mutually consistent.

▶ **Lemma 3.** *Consider an instance R of* `Read` *with arguments i and j, executed by p_u and let r be the executed by R on line 36. Let DT be the last instance of* `DynamicTraverse` *executed by p_u before R. Then, R returns as the weight for edge $e_{i,j}$ the value v, which is the weight written to $e_{i,j}$ by U, where U is the last instance of* `Update` *with arguments i, j, v, that was linearized before the linearization point of DT.*

We prove then that *d-traversals* have a linearization point inside their execution interval and use all the above to prove the following theorem.

▶ **Theorem 4.** `Dense` *is a wait-free linearizable concurrent graph implementation with $O(k)$ step complexity, where k is the number of active processes.*

6 Discussion

We have introduced a concurrent graph and provided an implementation, which supports wait-free operations, which include updates to the graph edges and the possibility of performing dynamically defined partial traversals. Operations implemented by `Dense` access and affect edges of a graph. Thus, the algorithm is designed with the implicit assumption of a fixed or at least, maximal number of possible vertices out of a specific vertex set. `Dense` operations are oblivious to the values or possible other attributes of those vertices. Indeed, there are many applications that are concerned with the connectivity of a graph only and need only access graph edges. Examples include garbage-collection – where objects are represented by graph nodes, while references to them are represented by graph edges – and graph-based video game navigation – where the edges of a graph represent walkable surfaces between obstacles, represented in turn by graph nodes. Nevertheless, an interesting line of future work is to extend the update and traversal capabilities of `Dense` to also provide information about the state or attributes of the visited vertices. `Dense` takes an irregular data structure and uses a regularized representation of it, in order to provide dynamic traversals. An interesting question concerns whether the helping mechanism employed by `Dense` can be used as a generalized traversal technique. It would be interesting to explore what other irregular or regular data structures (trees, lists, queues, etc) can benefit from it.

Acknowledgements. The authors would further like to thank Prof. Panagiota Fatourou for the useful comments and fruitful discussion that she provided. We thankfully acknowledge the support of the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project, and the support of the European Commission under the 7th Framework Programs through the EuroServer (FP7-ICT-610456) and HiPEAC3 (FP7-ICT-287759) projects.

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, September 1993. `doi:10.1145/153724.153741`.

2 Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *PVLDB*, 1(1):598–609, 2008.

3 Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015.

4 Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 336–343, NY, USA, 2008. ACM.

5 Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A dynamic elimination-combining stack algorithm. *CoRR*, abs/1106.6304, 2011.

6 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, NY, USA, 2014. ACM. doi:10.1145/2555243.2555267.

7 Vadim Bulitko, Yngvi Bjornsson, Nathan R. Sturtevant, and Ramon Lawrence. Real-time heuristic search for pathfinding in video games. In *Artificial Intelligence for Computer Games*, pages 1–30. Springer New York, 2011. doi:10.1007/978-1-4419-8188-2_1.

8 Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In *Proceedings of the 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 9–18, NY, USA, 2012. ACM.

9 Joseph Carsten, Arturo Rankin, Dave Ferguson, and Anthony Stentz. Global planning on the mars exploration rovers: Software integration and surface testing. *J. Field Robot.*, 26(4):337–357, April 2009. doi:10.1002/rob.v26:4.

10 Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 33rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 322–331, NY, USA, 2014. ACM. doi:10.1145/2611462.2611500.

11 Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *37th International Conference on Parallel Processing (ICPP)*, pages 536–545, 2008.

12 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, NY, USA, 2010. ACM. doi:10.1145/1835698.1835736.

13 Panagiota Fatourou, Mykhailo Iaremko, Eleni Kanellou, and Eleftherios Kosmas. Algorithmic techniques in stm design. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913, pages 101–126. Springer, 2015. doi:10.1007/978-3-319-14720-8_5.

14 Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, pages 1–46, 2013. doi:10.1007/s00224-013-9491-y.

15 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 300–314, London, UK, 2001. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645958.676105.

16 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing*, volume 6343, pages 79–93. Springer, 2010.

17 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215. ACM, 2004.

18 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

19 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

20 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

21 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.

22 Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. In *Distributed Computing*, volume 5805, pages 142–156. Springer Berlin Heidelberg, 2009.

23 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 408–419. Springer-Verlag, 2005. doi:10.1007/11590156_33.

24 Frank M. Johannes. Partitioning of vlsi circuits and systems. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC'96, pages 83–87, NY, USA, 1996. ACM.

25 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 223–234, NY, USA, 2011. ACM. doi:10.1145/1941553.1941585.

26 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, February 2012. doi:10.1145/2370036.2145835.

27 Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, December 1999. doi:10.1006/jpdc.1999.1578.

28 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, NY, USA, 1996. ACM. doi:10.1145/248052.248106.

29 Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 333–344, 2011. doi:10.1145/1950365.1950404.

30 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 239–248, 2015. doi:10.1109/IPDPS.2015.84.

31 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Distributed Computing*, volume 8205, pages 224–238. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-41527-2_16.

32 Aleksandar Prokopec, Nathan G. Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. *SIGPLAN Not.*, 47(8):151–160, Feb 2012. doi:10.1145/2370036.2145836.

33 Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. *SIGPLAN Not.*, 47(10):375–394, October 2012. doi:10.1145/2398857.2384644.

34 William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, NY, USA, 2006. ACM.

35 Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, NY, USA, 1995. ACM.

36 Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, November 2011. URL: `http://dl.acm.org/citation.cfm?id=1953048.2078187`.

37 S. Taylor, J.R. Watts, M.A. Rieffel, and M.E. Palmer. The concurrent graph: basic technology for irregular problems. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(2):15–25, Summer 1996. `doi:10.1109/88.494601`.

38 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 309–310, NY, USA, 2012. ACM. `doi:10.1145/2145816.2145869`.

39 Chung Yung, Jheng-Jyun Syu, and Shiang-Yu Yang. A graph-based algorithm of mostly incremental garbage collection for active object systems. In *International Computer Symposium (ICS)*, pages 988–996, 2010. `doi:10.1109/COMPSYM.2010.5685367`.

A Faster Counting Protocol for Anonymous Dynamic Networks

Alessia Milani[1] and Miguel A. Mosteiro[2]

1 **LABRI, University of Bordeaux 1, INP, Talence, France**
`milani@labri.fr`

2 **Department of Computer Science, Kean University, Union, USA**
`mmosteir@kean.edu`

Abstract

We study the problem of counting the number of nodes in a slotted-time communication network, under the challenging assumption that nodes do not have identifiers and the network topology changes frequently. That is, for each time slot links among nodes can change arbitrarily provided that the network is always connected.

This network model has been motivated by the ongoing development of new communication technologies that enable the deployment of a massive number of devices with highly dynamic connectivity patterns. Tolerating dynamic topologies is clearly crucial in face of mobility and unreliable communication. Current communication networks do have node identifiers though. Nevertheless, in future massive networks, it might be suitable to avoid nodes IDs to facilitate mass production. Consequently, knowing what is the cost of anonymity is of paramount importance to understand what is feasible or not for future generations of Dynamic Networks.

Counting is a fundamental task in distributed computing since knowing the size of the system often facilitates the desing of solutions for more complex problems. Also, the size of the system is usually used to decide termination in distributed algorithms. Currently, the best upper bound proved on the running time to compute the exact network size is double-exponential. However, only linear complexity lower bounds are known, leaving open the question of whether efficient Counting protocols for Anonymous Dynamic Networks exist or not.

In this paper we make a significant step towards answering this question by presenting a distributed Counting protocol for Anonymous Dynamic Networks which has exponential time complexity. This algorithm, which we call Incremental Counting, ensures that eventually every node knows the exact size of the system and stops executing the protocol. Previous Counting protocols have either double-exponential time complexity, or they are exponential but do not terminate, or terminate but do not provide running-time guarantees, or guarantee only an exponential upper bound on the network size. Other protocols are heuristic and do not guarantee the correct count.

1998 ACM Subject Classification F.2.0 General

Keywords and phrases Anonymous Dynamic Networks, Counting, Time-varying Graphs

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.28

1 Introduction

We study the problem of *Counting* the number of nodes in a communication network, under the challenging assumption that nodes do not have identifiers (IDs) and the network topology changes frequently. We consider broadcast networks in slotted-time scenarios. That is, in any given time slot, a message sent by a given node is received by all nodes directly connected to it (*one-hop neighbors*). Worst-case topology changes are modeled assuming the presence

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 28; pp. 28:1–28:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of an adversary that, for each time slot, chooses the set of links among nodes. The choice is arbitrary as long as, in each time slot, the network is connected. This dynamic topology model, called *1-interval connectivity*, was introduced in [10] for Dynamic Networks where each node has a unique identifier.

The network model described, called *Anonymous Dynamic Network*, has attracted a lot of attention recently [12, 4, 5, 6]. The model has been motivated by the ongoing development of new communication technologies that enable the deployment of a massive number of devices with highly dynamic connectivity patterns. Tolerating dynamic topologies is clearly crucial in face of mobility and unreliable communication. Current communication networks do have node IDs (or otherwise a labeling is defined at startup). Nevertheless, in future massive networks, it might be suitable to avoid nodes IDs to facilitate mass production. Consequently, knowing what is the cost of anonymity is of paramount importance to understand what is feasible or not for future generations of Dynamic Networks.

Counting is a fundamental distributed computing problem since knowing the size of the system facilitates the solution of more complex problems. Also this parameter is usually used to ensure the termination of the algorithm.

Counting can be solved in Anonymous Dynamic Networks, but the best known upper bound on the time complexity is double-exponential [4]. A double-exponential running time precludes the application of such algorithm to networks of significant size, but only linear lower bounds are known. Such a large gap leaves open the question of whether practical protocols exist or not.

The protocol presented in this paper makes a significant step towards answering the latter question, reducing the time complexity for *exact* Counting to exponential. Our algorithm, which we call Incremental Counting, ensures that there is a time slot when all nodes know the *exact size* of the system and they stop executing the algorithm. All nodes stop at the same round and this is known by every node. Thus it is easy to concatenate another algorithm which uses the system size.

Previous Counting protocols for Anonymous Dynamic Networks have either double-exponential time complexity [4], or they are exponential but do not terminate [4], or terminate but do not provide running-time guarantees [5], or guarantee only an exponential upper bound on the network size [12]. Other protocols are heuristic and do not guarantee the correct count [6].

All current Counting protocols for Anonymous Dynamic Networks [5, 12, 4, 6] assume the presence of one distinguished node, usually called *leader*, and additionally use some knowledge of the number of neighbors of each node, called *degree*. In our model, we include both assumptions. Namely, the presence of a leader node, and an upper bound on the maximum degree of the adversarial topology which is known by all nodes. While these assumptions may seem too strong it was proved in [12] that Counting is not solvable in Anonymous Networks without the presence of a *leader*, even if the topology does not change. In the same work, it was conjectured that any non-trivial computation is impossible without knowledge of some network characteristics.

Incremental Counting is inspired by the algorithm presented by Di Luna et al. in [4], which starts computing an upper bound on the network size using the algorithm presented in [12]. Then, it verifies each candidate size down to the correct size. To verify each candidate size, an energy-transfer approach is used. Namely, each non-leader node is initially assigned a unit of energy which is shared evenly with neighbors in each communication round, except for the leader that works as a sink. This energy-transfer protocol is a backwards version of *mass-distribution* and *gossip-based* algorithms [9, 1, 8] used to compute the size in other network models. The unit mass initially held in only one node in the latter system is shared throughout the network, converging to the average which is the inverse of the size. The energy-transfer protocol is shown to be at most exponential in the candidate size which in turn is exponential in the worst case, yielding a double-exponential Counting protocol.

Starting with an upper bound on the size of the system (as in previous works) facilitates the verification phase, because it tells the leader after how many rounds it has "heard" (i.e. received any needed information) from all the nodes. Unfortunately, it is not known how to obtain an upper bound better than exponential, and if an upper bound is not known, the challenge is to understand which is the condition the leader has to check to know that all nodes have been heard.

In this paper, Incremental Counting leverages the above idea of verifying candidate sizes using an energy-transfer protocol, but rather than starting with an upper bound, it follows a bottom-up approach. That is, it verifies $2, 3, \ldots,$ etc. up to the actual size. Then, an energy threshold is carefully chosen to decide when the count is accurate. This novel approach yields an exponential speedup in the worst-case running-time guarantees.

The running time proved also identifies the collection of energy at the leader as the speedup bottleneck for gossip-based Counting, given that all other factors in the time complexity obtained are polynomial. In contrast, in the running time of other exact Counting protocols that terminate, all factors are exponential or double exponential [4], or the running time is not proved [5].

Contributions

In the following we summarize the main contributions of our work.

- Following-up on the Conscious Counting protocol of [4], we present an improved Incremental Counting protocol for Anonymous Dynamic Networks that computes the exact number of nodes in less than $(2\Delta)^{n+1}(n+1)\ln(n+1)/\ln(2\Delta)$ communication rounds, where n is the number of nodes and Δ is any upper bound on the maximum number of neighbors that any node will ever have. Incremental Counting tolerates worst-case changes of topology, limited to 1-interval connectivity. The protocol requires the presence of one leader node and knowledge of Δ.
- The running time of Incremental Counting entails an exponential speedup over the previous best Counting algorithm in [4], which was proved to run in $O(e^{(\Delta^{2n})}\Delta^{3n})$ communication rounds, which is double-exponential. The speedup attained is mainly due to a carefully chosen energy threshold used to verify candidate sizes that are not bigger than the actual size. Our analysis shows the correctness of such verification.
- The time complexity proved identifies the phase where the leader collects energy from all other nodes as the speedup bottleneck for Counting with gossip-based protocols. Indeed, the exponential cost is due to this collection, whereas all other terms in the time complexity are polynomial. In contrast, in the running time of [4] all terms are exponential or double exponential.

Roadmap

The rest of the paper is organized as follows. In Section 2 we briefly overview previous work directly related to this paper. After formally defining the model and the problem in Section 3, we present the Incremental Counting protocol in Section 4 and its analysis in Section 5.

2 Related Work

The following is an overview of previous work on Counting in Anonymous Dynamic Networks directly related to this paper. Other related work may be found in a survey on Dynamic Networks and Time-varying Graphs by Casteigts et al. [3], and in the papers cited below.

Worst-case topology changes in Dynamic Networks may be limited assuming that the network is always connected (cf. [12, 10, 14, 4]), or sometimes disconnected but for some limited time (cf. [2, 15, 7, 13]). The *T-interval* connectivity model was introduced in [10]. For $T \geq 1$, a network is said to be T-interval connected if for every T consecutive rounds the network topology contains a stable connected subgraph spanning all nodes. In the same paper, a Counting protocol was presented, but it requires each node to have a unique identifier. In [10] it is also proved that, if no restriction on the size of the messages is required, the counting problem can be easily solved in $O(n)$ time when nodes have IDs. In our work, we focus on *Anonymous* Dynamic Networks. Understanding if a linear counting algorithm exists also when IDs are not available will help to understand the difficulty introduced by anonymity (if any).

A Counting protocol for Anonymous Dynamic Networks where an upper bound Δ on the maximum degree is known was presented in [12]. The adversarial topology is limited only to 1-interval connectivity, but the algorithm obtains only an upper bound on the size of the network n, which in the worst case is exponential, namely $O(\Delta^n)$. In our work, we aim to obtain an exact count, rather than only an upper bound.

The *Conscious Counting* algorithm presented later in [4] does obtain the exact count for the same network model, but requires knowledge of an initial upper bound K on the size of the network. Conscious Counting would be exponential if such upper bound were tight, since it runs in $O(e^{K^2}K^3)$ communication rounds. However, K is obtained using the algorithm in [12] mentioned above. Consequently, in the worst case the overall running time of the Conscious Counting Algorithm is $O(e^{(\Delta^{2n})}\Delta^{3n})$, which is double-exponential. In our work, we obtain the exact count in exponential time. That is, we reduce exponentially the best known upper bound for exact Counting.

Anonymous Dynamic Networks where an upper bound on the maximum degree is not known where also studied [4, 5, 6]. In [4], the protocol does not have a termination condition. That is, nodes running the protocol do not know whether the correct count has been reached or not. Hence, they have to continue running the protocol forever. In a companion paper [6], the authors stop the protocol heuristically. Hence, the count obtained is not guaranteed to be correct. Indeed, errors appear when the conductance of the underlying connectivity graph is low. In our work, we aim for Counting algorithms that terminate returning always the correct count. The protocol in [5] is shown to eventually terminate, although the running time is not proved. In their model, it is assumed that each node is equipped with an oracle that provides an estimation of its degree at each round. This is still an assumption of knowledge of network characteristics, although local. This and the above shortcomings are not unexpected in light of the conjecture in [12], which states that Counting (actually, any non-trivial computations) in Anonymous Dynamic Networks without knowledge of some network characteristics is impossible. Nevertheless, a proof of such conjecture has not been found yet.

Known lower bounds for Counting in Anonymous Dynamic Networks include only the trivial $\Omega(D)$, where D is the *dynamic* diameter of the network, and $\Omega(\log n)$ [1] even if D is constant, proved in [11].

[1] Throughout the paper, log means logarithm base 2, unless otherwise stated.

3 Preliminaries

3.1 The Counting Problem

An algorithm is said to solve the *Counting* problem if whenever it is executed in a Dynamic Network comprising n nodes, all nodes eventually terminate and output n.

3.2 The Anonymous Dynamic Network Model

We consider a synchronous Dynamic Network composed of a fixed set of nodes V where $|V| = n$. Nodes have no identifiers (IDs) or labels. We also assume the presence of a special node called the *leader* and denoted ℓ.

Nodes communicate by broadcast. In particular, communication proceeds in synchronous *rounds*. At each round a node broadcasts a message to its neighbors and simultaneously receives the messages broadcast in the same round by its neighbors (if any), then it makes some local computation. The time of computation is negligible. Thus, we compute the time complexity in rounds of communication.

At each round the set of communication links changes adversarially. Thus, the network is modeled as a dynamic graph $G = (V, E)$ where $E : \mathbb{N} \to \{(u, v) s.t. (u, v) \in V\}$ is a function mapping a round number r to a set of undirected edges $E(r)$. In particular, we consider the following 1-interval connectivity model proposed by Kuhn et al. in [10].

▶ **Definition 1.** A dynamic graph $G = (V, E)$ is 1-interval connected if for all $r \in \mathbb{N}$, the static graph $G_r := (V, E(r))$ is connected.

Finally, we assume that the size of the neighborhood of a node is upper bounded by a number $\Delta > 0$ at every round, and we assume that Δ is known by the nodes.

At a first glance, some knowledge of the degree seems unnecessary because, after one message from each neighbor has been received in a given round, the degree is simply the message count. However, for the next round of communication, the degree may change due to changing topology. Thus, a node does not know its current degree before sending messages to its neighbors.

4 Distributed Counting Algorithm

Incremental Counting consists of a sequence of iterations. In each iteration, a candidate size is checked to decide if it is correct. If not, the candidate size is increased and a new iteration starts. In the following, we provide a high level explanation of the algorithm executed in each iteration.

At the beginning of each iteration every node is assigned energy value 1, except for the leader which has 0 energy. Then, the iteration proceeds in three consecutive phases described below. Each phase lasts a fixed amount of rounds which only depends on the current estimation of the system size. This is intended to synchronize the computation at all the nodes in the system without extra communication.

During the first phase, called the *Collection Phase*, each node discharges itself by sending at each round a fraction at most half of its current energy to its neighbors. Then it computes its new energy by taking into account the energy given to its neighbors and the energy received from them. The leader acts as a sink collecting energy but not disseminating it. This phase completes when the leader has received an amount of energy such that, if the candidate size for the current iteration is the correct system size n, there is no node in the system with more than $1/k^c$ residual energy, for some constant $c > 1$. The function $\tau(k)$ in Algorithms 1 and 2 gives the number of iterations of the Collection Phase needed to guarantee this. An exponential upper bound on $\tau(k)$ is computed in Corollary 7. However, the bound may not be tight, so $\tau(k)$ is left as a parameter in the protocol. Should a better bound on $\tau(k)$ be proved, the protocol can be used as is.

Then, the *Verification Phase* starts. During this phase, the energy at each node does not change and the leader verifies the correctness of the current candidate size looking for a node with residual energy greater than $1/k^c$. To this aim at each round of the Verification phase each non leader node broadcasts the maximal energy it has "heard" during this phase. At the beginning each such node broadcasts its own residual energy. This phase lasts sufficiently long to ensure that if a node with residual energy greater than $1/k^c$ exists, then the leader will hear from it. If the leader does not hear from such node, it knows that the candidate size was indeed correct, and the verification phase completes successfully.

The last phase, called *Notification Phase*, is used by the leader when the verification phase completes successfully. To notify such event, the leader broadcasts a special $\langle Halt \rangle$ message, and each node in turn broadcasts it as soon as it is received and as long as the Notification Phase is not completed. If the Verification Phase completes unsuccessfully, the leader and every other node simply wait for the same number of rounds of communication without taking any action, and then all the nodes start a new iteration. This procedure ensures synchronism. A node stops executing the algorithm at the end of the Notification phase if it has received the $\langle Halt \rangle$ message. At this time every node knows the exact size of the system.

The Incremental Counting protocol for the leader and non-leader nodes is detailed in Algorithms 1 and 2.

PseudoCode

Variables at the leader node

- e_ℓ is the energy of the leader at the current round. It is initialized to 0 at the beginning of each iteration.
- k is the estimation of the system size. Initially equal to 1 and increased by one in each iteration.
- $1/k^c$ is a threshold value for the energy such that, for a given estimate k, if k is the correct size of the system, after the Collection Phase no node has energy greater than $1/k^c$ for some constant $c > 1$.
- $IsCorrect$, initially *true* is set to $false$ if the leader discovers that its estimate k is wrong. This happens if the value of $e_\ell > k - 1$ at the end of the Collection phase or if during the Verification phase the leader discovers a node with energy greater than $1/k^c$.
- $halt$, initially $false$ is set to *true* when the leader verifies that k is the correct size of the system.

Variables at non leader nodes

- e is the energy of the node at the current round. It is initialized to 1 at the beginning of each iteration.
- k is the estimation of the system size. Initially equal to 1 and increased by one in each iteration.
- e_{max}, is the maximum energy the node is aware of at the current round of the Verification Phase.
- $halt$, initially $false$, is set to *true* when the node receives a $\langle Halt \rangle$ message.

Algorithm 1: Incremental Counting algorithm for the leader node.

```
$k \leftarrow 1$
$halt \leftarrow false$
while $\neg halt$ do
    $k \leftarrow k + 1$
    $IsCorrect \leftarrow true$
    $e_\ell \leftarrow 0$
    // Collection Phase
    for each of $\tau(k)$ communication rounds do
        receive $e_1, e_2, \ldots e_s$ from neighbors, where $1 \leq s \leq \Delta$
        $e_\ell \leftarrow e_\ell + e_1 + e_2 + \ldots + e_s$
    // Verification Phase
    for each of $1 + \left\lceil \frac{k}{1-1/k^c} \right\rceil$ communication rounds do
        receive $e_1, e_2, \ldots e_s$ from neighbors, where $1 \leq s \leq \Delta$
        if $k - 1 - 1/k^c \leq e_\ell \leq k - 1$ then
            for $j := 1 \ldots s$ do
                if $e_j > 1/k^c$ then
                    $IsCorrect \leftarrow false$
        else
            $IsCorrect \leftarrow false$
    // Notification Phase
    for each of $k$ communication rounds do
        if $IsCorrect$ then
            broadcast $\langle Halt \rangle$
            $halt \leftarrow true$
        else
            do nothing
output $k$
```

5 Analysis

The following notation will be used. The energy of node i at the beginning of round r, is denoted as e_i^r, which is also generalized to any set of nodes $S \subseteq V$ as $e_S^r = \sum_{i \in S} e_i^r$. For any given round r and node i, let the set of neighbors of i be N_i^r and the average energy of i's neighbors be $\overline{e}_{N_i^r}$. The superindex indicating the round number will be omitted when clear from context or irrelevant. Also, at any time, let $\sum_{i \in V} e_i$ be called the *system energy* and $\sum_{i \in V \setminus \{\ell\}} e_i$ be called the *energy left*. At the beginning of each iteration of the protocol, that is, for each new size estimate k, the energy of the leader is reset to zero and the energy of the non-leader nodes is reset to 1. Thus, the system energy is $\sum_{i \in V} e_i = n - 1$ and the energy left is $\sum_{i \in V \setminus \{\ell\}} e_i = n - 1$.

▶ **Lemma 2.** *For any network of n nodes, including a leader ℓ, running the* Incremental Counting *Protocol under the communication and connectivity models defined the following holds. For any given node $i \in V \setminus \{\ell\}$ and for any given round r of the Collection Phase, it is $e_i^r \leq 1$.*

Algorithm 2: Incremental Counting algorithm for non-leader nodes.

```
$k \leftarrow 1$
$halt \leftarrow false$
while $\neg halt$ do
    $k \leftarrow k + 1$
    $e \leftarrow 1$
    // Collection Phase
    for each of $\tau(k)$ communication rounds do
        broadcast $\langle \frac{e}{2\Delta} \rangle$ and receive $e_1, e_2, \ldots e_s$ from neighbors, where $1 \leq s \leq \Delta$
        $e \leftarrow e \cdot (1 - \frac{s}{2\Delta}) + \sum_{j=1}^{s} e_j$
    // Verification Phase
    $e_{max} \leftarrow e$
    for each of $\left\lceil 1 + \frac{k}{1-1/k^c} \right\rceil$ communication rounds do
        broadcast $\langle e_{max} \rangle$ and receive $e_1, e_2, \ldots e_s$ from neighbors, where $1 \leq s \leq \Delta$
        for $j := 1 \ldots s$ do
            if $e_j > e_{max}$ then
                $e_{max} \leftarrow e_j$
    // Notification Phase
    for each of $k$ communication rounds do
        if $halt$ then
            broadcast $\langle Halt \rangle$
        if receive $\langle Halt \rangle$ from some neighbor
         then
            $halt \leftarrow true$
output $k$
```

Proof. Fix some arbitrary (non-leader) node i. Consider the transition between round r and $r+1$. We have that

$$e_i^{r+1} \leq e_i^r + \overline{e}_{N_i^r} \frac{|N_i^r|}{2\Delta} - e_i^r \frac{|N_i^r|}{2\Delta} = e_i^r + (\overline{e}_{N_i^r} - e_i^r) \frac{|N_i^r|}{2\Delta}.$$

If $\overline{e}_{N_i^r} \leq e_i^r$, then $e_i^{r+1} \leq e_i^r$. That is, i's energy does not increase from round r to round $r+1$. If on the other hand it is $\overline{e}_{N_i^r} > e_i^r$, we have

$$e_i^{r+1} \leq e_i^r + (\overline{e}_{N_i^r} - e_i^r)/2 = (e_i^r + \overline{e}_{N_i^r})/2.$$

That is, the energy of i in round $r+1$ is at most the average between the energy of i in round r and the average of i's neighbors' energy in round r.

Now consider the evolution of the protocol along many rounds. We ignore the rounds when $\overline{e}_{N_i^r} \leq e_i^r$ since they do not increase the energy. For the other rounds, given that all nodes start with energy 1, and that the average of some numbers cannot be bigger than the maximum, the energy at any given node cannot get bigger than 1. Hence, the claim follows. ◀

▶ **Lemma 3.** *For any network of n nodes, under the communication and connectivity models defined, the following holds. If a message m is held by all nodes in a set $V_1 \subseteq V$, after*

$|V| - |V_1|$ rounds when every node holding the message broadcasts m in each round, all nodes in V hold the message.

Proof. For any round $r > 0$, consider the partition of nodes $\{V_1^r, V_2^r\}$ defined by the nodes holding the message at the beginning of round r. That is, $\forall i \in V_1^r$ the node i holds m and $\forall j \in V_2^r$ the node j does not hold m. By 1-interval connectivity, there must exist a link u, v, such that $u \in V_1^r$ and $v \in V_2^r$. Given that all nodes holding the message broadcast m, v must receive the message in round r. Thus, at the beginning of round $r+1$ it is $|V_1^{r+1}| \geq |V_1^r| + 1$ and $|V_2^{r+1}| \leq |V_2^r| - 1$. Applying the same argument inductively, after $|V_2^{r+1}|$ more rounds all nodes hold the message. ◀

The following lemma is a straightforward application of Lemma 3 to the Notification Phase, where the message broadcasted is $\langle Halt \rangle$ for the first time when $k = n$.

▶ **Lemma 4** (Correctness of the Notification Phase). *For any network of n nodes, including a leader ℓ, running the* Incremental Counting *Protocol under the communication and connectivity models defined the following holds. If at the end of the Verification Phase $IsCorrect = true$, then at the end of the Notification Phase all nodes stop the Counting Protocol holding the size n.*

▶ **Lemma 5** (Correctness of the Verification Phase). *For any network of $n > 3$ nodes, including a leader ℓ, running the* Incremental Counting *Protocol under the communication and connectivity models defined the following holds. For any estimate of the size of the network k and constant $c > 1$, at the end of the Verification Phase $IsCorrect = true$ if and only if $k = n$.*

Proof. We start observing that, for each estimate k, each non-leader node is initialized with one unit of energy (Line 5 in Algorithm 2) and the leader's energy is initialized to 0 (Line 6 in Algorithm 1). Until a new iteration of the outer loop (in both algorithms) is executed, no energy is lost or gained by the system as a whole. Hence, the system energy is always $n - 1$.

We prove first that, if $k = n$, at the end of the Verification Phase it is $IsCorrect = true$. Given that $k = n$, the system energy is $k - 1$ and therefore $e_\ell \leq k - 1$. Also because $k = n$, we know that after the Collection Phase it is $e_\ell \geq k - 1 - 1/k^c$ by definition of $\tau(k)$. Therefore, $IsCorrect$ is not set to false in Line 17 of Algorithm 1. Also because $e_\ell \geq k - 1 - 1/k^c$ at the end of the Collection Phase, we know that the energy left at the beginning of the Verification Phase is $e_{V \setminus \{\ell\}} = k - 1 - e_\ell \leq 1/k^c$. Therefore, no non-leader node could have more than that energy. That is, $\forall i \in V \setminus \{\ell\} : e_i \leq 1/k^c$. Thus, during the Verification Phase, the leader will not be able to detect a node with energy bigger than $1/k^c$. Therefore, $IsCorrect$ is not set to false in Line 15 of Algorithm 1 either. There is no other line where $IsCorrect$ is set to false. Hence, at the end of the Verification Phase it is $IsCorrect = true$.

We prove now the other direction of the implication. That is, if at the end of the Verification Phase $IsCorrect = true$, then it is $k = n$. For the sake of contradiction, assume that $IsCorrect = true$ but $k \neq n$. Notice that k cannot be larger than n, because the estimate is increased one by one, we already proved that if $k = n$ at the end of the Verification Phase it is $IsCorrect = true$, and Lemma 4 shows that all nodes would have stopped running the protocol. Thus, we are left with the case when $k < n$.

Notice that if $e_\ell > k - 1$ the variable $IsCorrect$ is set to false in Line 17 of Algorithm 1. Hence, it must be $e_\ell \leq k - 1$ and, given that the system energy is $n - 1$, the energy left is $e_{V \setminus \{\ell\}} \geq n - k$. This energy left is stored in the $n - 1$ non-leader nodes. Hence, there must exist some node $j \in V \setminus \{\ell\}$ in the network such that $e_j \geq (n-k)/(n-1)$. If $IsCorrect = true$ it means that the leader did not detect a node with energy bigger than $1/k^c$ in Line 14 of

Algorithm 1. However, for any $2 \leq k \leq n-1$, $n > 3$, and $c > 1$, it is $1/k^c < (n-k)/(n-1)$ which means that such node must exist.

To see why the latter inequality is true, we verify that $k^c(n-k) - n + 1 > 0$ as follows. With respect to k, this function has a maximum for $k = cn/(c+1)$. That is, for $2 \leq k \leq n-1$ (recall that we are in the case $k < n$), the function has minima in 2 and $n-1$. Then, it is enough to verify that $2^c(n-2) - n + 1 > 0$, which is true for any $c > 1$ and $n > 3$, and that $(n-1)^c - n + 1 > 0$, which is also true for any $c > 1$ and $n \geq 2$.

Thus, to complete the proof, it is enough to show that $1 + k^{c+1}/(k^c - 1)$ rounds are enough to detect a node with energy bigger than $1/k^c$. To do that, we upper bound the number of nodes in the network with energy at most $1/k^c$ as follows. We know that at any time when the leader has energy e_ℓ, the energy left is $n - 1 - e_\ell$. Let $S \subseteq V$ be the set of nodes with energy at most $1/k^c$. Then, we have that $n - 1 - e_\ell = \sum_{j \in S} e_j + \sum_{k \in V \setminus S} e_k$. To maximize the size of S, we minimize the size of $V \setminus S$ assuming that all nodes in $V \setminus S$ have maximum energy, which according to Lemma 2 is at most 1. Then, we have that $n - 1 - e_\ell = \sum_{j \in S} e_j + (n - |S|)$ which yields $|S| - 1 - e_\ell = \sum_{j \in S} e_j$ Given that $\sum_{j \in S} e_j \leq |S|/k^c$, we have that $|S| \leq (1 + e_\ell)/(1 - 1/k^c)$. Recall that $e_\ell \leq k - 1$ because *IsCorrect* would have been set to false in Line 17 of Algorithm 1 otherwise. Replacing, we get $|S| \leq k^{c+1}/(k^c - 1)$.

Let $\{V_1, V_2\}$ be a partition of V such that $V_2 = S \cup \{\ell\}$. Recall that, for any $v \in V_1$ it is $e_v > 1/k^c$. Using Lemma 3, we know that $|V_2| = 1 + k^{c+1}/(k^c - 1)$ iterations in the Verification Phase of Algorithm 1 are enough for the leader to detect that there is a node with energy larger than $1/k^c$, which contradicts our assumption that $IsCorrect = true$. ◀

The following theorem establishes our main result.

▶ **Theorem 6.** *For any anonymous dynamic network of $n > 3$ nodes, including a leader ℓ, and for any constant $c > \log 5$, the following holds. If the adversarial topology is limited by a maximum degree Δ and the connectivity model defined, and nodes run the* Incremental Counting *Protocol in Algorithms 1 and 2 under the communication model defined, after r rounds, all nodes stop holding the size of the network n, where*

$$r < n(n+3) + \ln n - 4 + \sum_{k=2}^{n} \tau(k).$$

Where $\tau(k)$ is a function such that, if $k = n$ and the Collection Phase is executed for at least $\tau(k)$ rounds, then at the end of the phase the leader has energy $e_\ell \geq k - 1 - 1/k^c$.

Proof. Correctness is a direct consequence of Lemmas 4 and 5. The running time is obtained adding the number of iterations of each phase, as follows.

$$\begin{aligned} r &= \sum_{k=2}^{n} \left(\tau(k) + \left\lceil 1 + \frac{k}{1 - 1/k^c} \right\rceil + k \right) \\ &\leq \sum_{k=2}^{n} \left(\tau(k) + 2 + \frac{k}{1 - 1/k^c} + k \right) \\ &= n(n+3) - 4 + \sum_{k=2}^{n} \left(\tau(k) + \frac{k}{k^c - 1} \right). \end{aligned}$$

Using that $k/(k^c - 1) < 1/k$ for any $c > \log 5$ and $k \geq 2$, we obtain the following.

$$\begin{aligned} r &< n(n+3) - 4 + \sum_{k=2}^{n} \left(\tau(k) + \frac{1}{k} \right) \\ &\leq n(n+3) + \ln n - 4 + \sum_{k=2}^{n} \tau(k). \end{aligned}$$ ◀

Bounding the running time of the Collection Phase using Lemma 2 in [4] in Theorem 6, the following corollary is obtained.

▶ **Corollary 7.** *For any anonymous dynamic network of $n > 6$ nodes, including a leader ℓ, the following holds. If the adversarial topology is limited by a maximum degree $1 \leq \Delta \leq n-1$ and the connectivity model defined, and nodes run the* Incremental Counting *Protocol in Algorithms 1 and 2 under the communication model defined, after r rounds, all nodes stop holding the size of the network n, where*

$$r < \frac{(2\Delta)^{n+1}(n+1)\ln(n+1)}{\ln(2\Delta)}.$$

Proof. Lemma 2 in [4] proves that, for any estimate $k \geq n$ and integer $\rho > 0$, starting with $e_\ell = 0$ and $e_i = 1$ for all $i \in V \setminus \{\ell\}$, after running ρk rounds of the energy transfer protocol the energy stored in the leader is $e_\ell \geq n(1 - (((2\Delta)^k - 1)/(2\Delta)^k)^\rho)$. Notice in Theorem 6 that the condition $e_\ell \geq k - 1 - 1/k^c$ only applies when $k = n$. Thus, it is enough to find ρ such that

$$\begin{aligned} k \left(1 - \left(\frac{(2\Delta)^k - 1}{(2\Delta)^k} \right)^\rho \right) &\geq k - 1 - 1/k^c \\ \rho &\geq \frac{\ln(k/(1 + 1/k^c))}{\ln\left(1/(1 - 1/(2\Delta)^k)\right)}. \end{aligned}$$

Using that $1 - x \leq e^{-x}$ for $x \leq 1$, it is enough to have $\rho = \lceil (2\Delta)^k \ln k \rceil$. Replacing in Theorem 6, we obtain

$$\begin{aligned} r &< n(n+3) + \ln n - 4 + \sum_{k=2}^{n} k \lceil (2\Delta)^k \ln k \rceil \\ &\leq n(n+3) + \ln n - 4 + \sum_{k=2}^{n} k(1 + (2\Delta)^k \ln k) \\ &= n(3n+7)/2 + \ln n - 5 + \sum_{k=2}^{n} k(2\Delta)^k \ln k. \end{aligned}$$

Bounding with the integral,

$$\begin{aligned} r &< n(3n+7)/2 + \ln n - 5 + \int_{k=2}^{n+1} k(2\Delta)^k \ln k \; \mathrm{d}k \\ &= n(3n+7)/2 + \ln n - 5 + \left. \frac{(2\Delta)^k((k\ln(2\Delta) - 1)\ln k - 1) + \mathrm{Ei}(k\ln(2\Delta))}{\ln^2(2\Delta)} \right|_2^{n+1} \\ &\leq n(3n+7)/2 + \ln n + \\ &\qquad \frac{(2\Delta)^{n+1}(((n+1)\ln(2\Delta) - 1)\ln(n+1) - 1) + \mathrm{Ei}((n+1)\ln(2\Delta))}{\ln^2(2\Delta)}. \end{aligned}$$

Using that $\mathrm{Ei}(\ln x) = \mathrm{li}(x) < x$, for any real number $x \neq 1$, it is $\mathrm{Ei}((n+1)\ln(2\Delta)) < (2\Delta)^{n+1}$. Replacing,

$$\begin{aligned} r &< n(3n+7)/2 + \ln n + \frac{(2\Delta)^{n+1}((n+1)\ln(2\Delta)-1)\ln(n+1)}{\ln^2(2\Delta)} \\ &= n(3n+7)/2 + \ln n + \frac{(2\Delta)^{n+1}(n+1)\ln(n+1)}{\ln(2\Delta)} - \frac{(2\Delta)^{n+1}\ln(n+1)}{\ln^2(2\Delta)}. \end{aligned}$$

Using that $n(3n+7)/2 + \ln n < (2\Delta)^{n+1}\ln(n+1)/\ln^2(2\Delta)$ for any $n > 6$ and $1 \leq \Delta \leq n-1$, the claim follows. ◀

5.1 Discussion

In this paper we have studied the problem of Counting in Anonymous Dynamic Networks. The problem is challenging because the lack of identifiers and changing topology make difficult to decide if a new message has been received before from the same node. Also, the obvious lack of knowledge of the network size makes difficult to decide when the algorithm has to stop.

Assuming an upper bound on the size of the system facilitates termination but may lead to very bad time complexity if the upper bound is a huge overestimate. According to our knowledge, the algorithm in [12] is the only one to compute an upper bound of the system size for Anonymous Dynamic Networks and in the worst case it is exponential, i.e. $O(\Delta^n)$ where n is the size of the system and Δ is an upper bound on the nodes' degree. Finding the termination condition when an upper bound on the network size is not available is more challenging, but it is expected to provide more efficient algorithms. Our Incremental Counting algorithm does not assume such upper bound, and computes the exact size of the system applying a bottom-up approach where the size is possibly underestimated several times.

It is known that if no restriction on the size of the messages is required, the Counting problem can be easily solved in $O(n)$ time when nodes have IDs [10]). In this paper, we have made a significant step towards understanding if a linear Counting algorithm exists also when IDs are not available, by identifying the speedup bottleneck and reducing exponentially the best known upper bound. This will help to understand the difficulty introduced by anonymity (if any). Despite our contribution, there is still a big gap with respect to the linear lower bound trivially given by the dynamic diameter.

Finally, although we focus on communication networks, our results carry over into any distributed system of similar characteristics.

Acknowledgements. We thank Arnaud Casteigts for introducing the model to us, and Antonio Fernández Anta for useful discussions.

References

1 Paulo Sérgio Almeida, Carlos Baquero, Martín Farach-Colton, Paulo Jesus, and Miguel A. Mosteiro. Fault-tolerant aggregation: Flow-updating meets mass-distribution. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems – 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2011. doi:10.1007/978-3-642-25873-2_35.

2 Antonio Fernández Anta, Alessia Milani, Miguel A. Mosteiro, and Shmuel Zaks. Opportunistic information dissemination in mobile ad-hoc networks: the profit of global synchrony. *Distributed Computing*, 25(4):279–296, 2012. doi:10.1007/s00446-012-0165-9.

3 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

4 Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Conscious and unconscious counting on anonymous dynamic networks. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 257–271. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-45249-9_17.

5 Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. Counting in anonymous dynamic networks under worst-case adversary. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 338–347. IEEE, 2014.

6 Giuseppe Antonio Di Luna, Silvia Bonomi, Ioannis Chatzigiannakis, and Roberto Baldoni. Counting in anonymous dynamic networks: An experimental perspective. In Paola Flocchini, Jie Gao, Evangelos Kranakis, and Friedhelm Meyer auf der Heide, editors, *Algorithms for Sensor Systems*, volume 8243 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-45346-5_11.

7 K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 27–34, 2003.

8 Antonio Fernández Anta, Miguel A. Mosteiro, and Christopher 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.

9 D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proc. of the 44th IEEE Ann. Symp. on Foundations of Computer Science*, pages 482–491, 2003.

10 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC'10, pages 513–522, New York, NY, USA, 2010. ACM. doi:10.1145/1806689.1806760.

11 Giuseppe Antonio Di Luna and Roberto Baldoni. Investigating the cost of anonymity on dynamic networks. *CoRR*, abs/1505.03509, 2015. URL: http://arxiv.org/abs/1505.03509.

12 Othon Michail, Ioannis Chatzigiannakis, and Paul G Spirakis. Naming and counting in anonymous unknown dynamic networks. In *Stabilization, Safety, and Security of Distributed Systems*, pages 281–295. Springer, 2013.

13 Othon Michail, Ioannis Chatzigiannakis, and Paul G Spirakis. Causality, influence, and computation in possibly disconnected synchronous dynamic networks. *Journal of Parallel and Distributed Computing*, 74(1):2016–2026, 2014.

14 Regina O'Dell and Rogert Wattenhofer. Information dissemination in highly dynamic graphs. In *Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing*, DIALM-POMC'05, pages 104–110, New York, NY, USA, 2005. ACM. doi:10.1145/1080810.1080828.

15 L. Pelusi, A. Passarella, and M. Conti. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *Communications Magazine, IEEE*, 44(11):134–141, 2006.

ActiveMonitor: Asynchronous Monitor Framework for Scalability and Multi-Object Synchronization*

Wei-Lun Hung[1], Himanshu Chauhan[2], and Vijay K. Garg[3]

1 **University of Texas, Austin, USA**
`wlhung@utexas.edu`
2 **University of Texas, Austin, USA**
`himanshu@utexas.edu`
3 **University of Texas, Austin, USA**
`garg@ece.utexas.edu`

Abstract

Monitor objects are used extensively for thread-safety and synchronization in shared memory parallel programs. They provide ease of use, and enable straightforward correctness analysis. However, they inhibit parallelism by enforcing serial executions of critical sections, and thus the performance of parallel programs with monitors scales poorly with number of processes. Their current design and implementation is also ill-suited for thread synchronization across multiple thread-safe objects. We present ActiveMonitor – a framework that allows multi-object synchronization without global locks, and improves parallelism by exploiting asynchronous execution of critical sections. We evaluate the performance of Java based implementation of ActiveMonitor on micro-benchmarks involving light and heavy critical sections, as well as on single-source-shortest-path problem in directed graphs. Our results show that on most of these problems, ActiveMonitor based programs outperform programs implemented using Java's reentrant-lock and condition constructs.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrent/parallel programming, monitors, concurrency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.29

1 Introduction

Most, if not all, programmers follow a standard recipe to implement shared memory parallel programs: they identify the critical sections in the serial implementation of the program, and make them thread-safe in the style of monitors [22]. Monitors provide dual abstractions: mutual exclusion and synchronization between threads. Their simplicity and elegance of use, and ready availability of mutexes/locks are two key factors behind such a wide adoption of this style. By enforcing serialized executions of critical sections, mutexes trivially guarantee the safety of data. Under high contention scenarios, however, such serialized executions become obvious performance bottleneck. In addition, mutexes force memory fencing due to which latency hiding techniques such as caching, pre-fetching, and operation re-ordering cannot be exploited to their fullest. As a combined effect of all these factors, programs in traditional monitor-style fare poorly in terms of throughput and scalability on multi-core CPUs. Mutex-based monitor implementations have another limitation: method invocations

* Supported in part by NSF Grants CNS-1346245, CNS-1115808, and Cullen Trust. A short version of this paper appeared as a brief announcement in DISC'2014.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 29; pp. 29:1–29:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

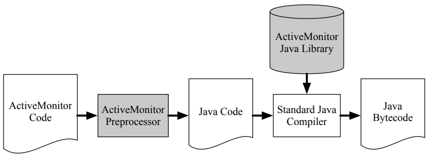


Figure 1 ActiveMonitor framework.

across multiple monitors cannot be combined easily. For example, given two thread-safe blocking queues, consider the problem of dequeueing an item from either of them. There is no easy solution to the problem of using mutex based synchronous monitors [19].

We present ActiveMonitor, a framework that provides significant programming ease in writing thread-safe programs, allows multi-object synchronization, as well as improves the runtime performance of these programs by exploiting asynchronous delegated executions on modern multi-core hardware. Extending our previous work AutoSynch [24], which provides waituntil keyword for automatic signaling and thread synchronization, ActiveMonitor framework enables asynchronous executions of critical sections, as well as method composition across monitor objects through simple constructs. Recall that monitors were envisioned in 1970's when saving processor cycles of the single-core CPUs was a primary programming concern. In contrast, not only multi-core processors are now ubiquitous, but they are also significantly cheaper and faster. In order to exploit the multi-core resources, we allow a monitor object to exist as a thread – hence it becomes an *active* artifact of the program. With this change, method invocations on this monitor object can be delegated [35]. In addition, we allow the monitor thread to execute critical sections *asynchronously*, so that calling threads can return to their local work without waiting for their completion.

Using ActiveMonitor involves the following steps (Fig. 1 shows the framework overview):

(a) The programmer writes a monitor based parallel program using the ActiveMonitor keywords. These keywords are: monitor, waituntil, synchronous, asynchronous, and notthreadsafe. He/she can use two additional operators OR and AND for compositionality across multiple monitor objects. ActiveMonitor automatically manages the use use of locks, and their acquisition/release so that the user is not required to explicitly program them. The user is also free from the responsibility of checking the predicate condition(s) and signaling appropriate threads. The framework observes the values of predicate conditions at runtime, and signals the appropriate threads automatically.

(b) He/she then runs the ActiveMonitor pre-processor to generate the program's equivalent Java code. The pre-processor injects code snippets to provide the corresponding functionality of framework keywords. The pre-processor also links invocations of ActiveMonitor runtime library API in the generated code.

(c) The program is then compiled as a standard Java program, and the binaries benefit from asynchronous executions of critical sections, and automatic signaling. If needed, the user can easily disable asynchronous executions at runtime by simply passing a flag.

ActiveMonitor enables operations that are not possible with traditional synchronous monitors. Solving the problem of removing an element from either of n blocking queues, where $n \geq 2$, is a challenging task with traditional monitors [19]. In ActiveMonitor it is just a matter of using the framework's OR construct: x = Q1.deqeue() OR Q2.dequeue() .... Similarly, the AND construct of the framework allows the programmer to aggregate results from multiple operations across different monitors. Our design and implementation integrates seamlessly with current constructs provided by most programming languages, and can thus benefit existing programs with only a handful of syntactic changes. The results of our experimental evaluation (using Java[1]) on five multi-threading problems show that ActiveMonitor outperforms, by a factor of two or more in some cases, traditional monitor based programs implemented using Java's ReentrantLock [30], and delegation technique [35] on most of these problems. In our current implementation of ActiveMonitor, use of thread dependent variables and functions is restricted. Note that this only disables the asynchronous executions provided by ActiveMonitor and the framework can still be used for such problems. We discuss these issues in § 9.

2 ActiveMonitor: Concepts & Design

In ActiveMonitor framework, each method of a monitor is a critical section – unless otherwise specified (using notthreadsafe keyword described shortly ahead). We use the term *worker* to denote an application thread/process. A monitor object can be instantiated as a thread/process based on the availability of system resources. This thread is called a *server*, and invocation of critical sections of monitor by workers are delegated to it. Delegation [35] is a technique in which critical sections of a monitor are not executed directly by workers invoking the method, but are processed by the *server* thread on behalf of workers. The workers *announce* their execution requests – in the form of *tasks* – to the server by adding the requests (task objects) to a shared storage that is owned by the monitor. Combining [15, 10] is a version of delegation in which the role of server is assumed by the worker that succeeds in acquiring the lock to the critical section. This thread becomes the *combiner*, and in addition to its own request, serves requests announced by other threads for a period of time before releasing the lock and allowing some other thread to become the combiner. Throughout this paper, we use the term *server* in both delegation and combining contexts. A critical section is *asynchronous* (or non-blocking) if the worker can return to executing its own local program from the critical section before its completion. Otherwise the critical section is synchronous (or blocking).

ActiveMonitor provides the following constructs for writing monitor based programs:

1. monitor: keyword that declares a class as a monitor, and frees the user from explicit lock instantiations, and their acquisition/release to make the critical sections thread-safe.
2. waituntil: a statement for conditional waits and notifications. The statement requires a boolean predicate as an argument.
3. synchronous: keyword used in declaration of monitor methods. Such methods are made thread-safe but not delegated to the server (monitor thread) for execution.
4. asynchronous: keyword used in declaration of monitor methods. Such methods are delegated to the server (monitor thread) , and the worker thread returns to its own local execution before completing the method. If the worker requires the result of the computation, it receives a future [12] instance which can be evaluated – a blocking call if the result is not yet available – to fetch the result.
5. notthreadsafe: this keyword in a method signature tells the framework to not generate thread-safe code for this method. incompatible with the previous two keywords: waituntil and asynchronous.
6. OR/AND: operators for logical composition of monitor methods. If a result is required from either of these operator calls, then the framework stipulates that all the operand

[1] Our technique is not limited to Java, and applies to any other modern programming language.

```
monitor class BoundedQueue<T> {
  T[] items;
  int putPtr, takePtr, count, size;
  BoundedQueue(int size) {
    this.size = size;
    items = new Object[size];
  }
  aysnchronous void put(T item) {
    waituntil(count < size);
    items [putPtr++] = item;
    putPtr = putPtr % size;
    ++count;
  }
  T take() {
    waituntil(count > 0);
    T x = (T)(items [takePtr++]);
    takePtr = takePtr % size;
    --count;
    return x;
  }
}
```

Figure 2 Bounded-Queue with ActiveMonitor.

method calls have the same return type. The order of operations is defined based on the evaluation of the pre-conditions (of operand monitor methods) at runtime.

Defaults: ActiveMonitor makes all monitor methods thread-safe by default. Each method that returns void and updates monitor state is asynchronous by default unless otherwise declared. Each method that returns a type value (and not a void) is made synchronous unless explicitly declared asynchronous by the user. Each read-only method – determined by static analysis of the program in the pre-processing/compilation phase – is also made synchronous irrespective of its return type. By doing so, the framework is able to use read-locks for such methods to exploit the inherent read parallelism in the program. The bounded queue implementation in Fig. 2 shows the actual usage of monitor, and asynchronous keywords, as well as the waituntil statement. Note that take() method will be made synchronous by the framework as it returns a value and is not explicitly declared asynchronous. As shown in the design overview of Fig. 1, the framework has two main components: a pre-processor and a runtime Java library. The pre-processor translates ActiveMonitor code into Java code. In addition, it also identifies the critical sections that are eligible for asynchronous execution. For each such method (critical section), the pre-processor generates its equivalent *task*.

It then replaces invocation of these methods (by application threads on monitor object) by submission of tasks to the server of the monitor. The runtime library has two sub-components: condition manager and task executer. The condition manager is responsible for observing the state of the monitor object for conditional waits and signaling an appropriate thread whenever its precondition becomes true. The task executer component manages the submission and completion of monitor tasks and also handles their asynchronous executions.

Our pre-processor uses a set of parsing rules that identify the ActiveMonitor keywords, and is an extension of the pre-processor in our previous work AutoSynch [24]. We briefly discuss its steps, and refer the reader to [24] for details. For a source class that is declared monitor, the pre-processor ensures that each method of the class is protected using the re-entrant lock by inserting lock acquisition and release statements at the beginning and end of method code. It then parses the method code for waituntil statements, and for each such statement it creates a new condition in the monitor class. For every condition, the notification criteria is the boolean predicate provided as the argument to its corresponding waituntil statement. Then it analyzes the method to decide whether or not it should be delegated. If the method is declared asynchronous or does not returns a value and updates the shared data, the pre-processor generates an equivalent task for delegation. We discuss monitor tasks, their generation and compositionality in the next section.

3 Monitor Tasks

In ActiveMonitor, a monitor task is defined as follows.

▶ **Definition 1** (Monitor Task)**.** A monitor task t consists of a boolean predicate P and a set of statements S. At runtime, if the precondition defined by P is true then t is 'executable' and statements in S can be executed to complete t. Otherwise, t is 'unexecutable'.

For a task t, its set of statements S can be empty. The pre-condition P – passed as an argument to waituntil statement – can either be absent altogether or may not appear as the first statement in the monitor method. When a monitor method has no precondition, the pre-processor creates a task with its precondition as tautology, indicating that the task can be executed at any time. If a monitor method does not start with a waituntil statement but has some such statement in between, then the precondition of the first derived task is a tautology. Consider the put method (lines 8–13) of the bounded-buffer program of Fig. 2. For this monitor method, the equivalent monitor task t is defined by the code of lines 9–12. For t, the precondition P is (count $<$ buffer_size); and it checks if the buffer has any space to insert the item. If this condition is false, the waituntil construct ensures that any thread trying to complete this task has to wait until the buffer has some space to insert the items. Lines 10 and 11 together form the set of statements S. The method is explicitly declared asynchronous, so the generated task is submitted for an asynchronous execution to the monitor thread.

3.1 Asynchronous Execution of Tasks

After an equivalent task t for a method m has been generated, all the invocations of m by workers are executed with combining technique [15, 10]. We use *futures* [12] for asynchronous (non-blocking) executions of critical sections. For each asynchronous method call the pre-processing phase injects submission of a task to the server (monitor thread) . A *future* reference is returned to the worker as a pointer to the computation. Whenever the server finishes the execution of a task, it updates its corresponding future reference with the result of the computation. If the worker needs the result of the computation it *evaluates* the future. Evaluation of a future is a blocking method: if the computation has not finished then the caller must wait until its completion. Note that unlike the schemes of [35, 15, 10], neither the server nor the worker threads perform busy-wait/spinning in ActiveMonitor. Thus, we do not waste any processing cycles and yield the CPU when there are no tasks to execute. Hence, ActiveMonitor provides a much more practical implementation for delegated executions.

To guarantee program order, ActiveMonitor framework stipulates that each worker can only submit one asynchronous task at a time. The task executor sub-component of the runtime library handles this by storing a map of ids of worker threads and their corresponding task submissions. Whenever a worker tries to submit an asynchronous task, it first checks the map to verify if there is some previous asynchronous task stored against its id that is not yet finished. The worker is forced to wait – by evaluating the future – for the completion of that task before being allowed to submit the new task. If the programmer understands the implications of out-of-program-order asynchronous executions, and wishes to exploit them then he/she can relax the program order execution by passing an argument to the runtime library. This change usually results in higher program throughputs. A detailed discussion on this topic can be found in our technical report [1].

4 Runtime Library

The runtime library of ActiveMonitor provides two key functionalities: (a) automatic signaling of threads under conditional waiting, and (b) delegation and asynchronous executions of critical sections. We extend our previous work AutoSynch [24] to enable functionality (a) for task based asynchronous executions and for multi-object synchronization through OR/AND operators. We summarize the key concepts here, and refer the interested reader to [24] for details.

4.1 Automatic Signaling

In current programming languages/libraries conditional synchronization through mutexes requires programmers to explicitly associate conditional predicates with condition variables and call *signal* (*signalAll*) or *await* statements manually. In contrast, ActiveMonitor framework manages conditional synchronization and thread signaling, and relieves the programmer of their explicit handling. The programmer only needs to use the waituntil clause. The idea of automatic signaling was initially explored by Hoare [22], but rejected in favor of condition variables due to efficiency considerations. Buhr et al. [3] claim that automatic monitors are 10 to 50 times slower than explicit signals. This is mainly due the sub-optimal implementation techniques that result in excessive predicate evaluations for conditions and subsequent context switches. In [24], we provide an efficient mechanism that improves the automatic signaling performance tremendously.

We use three concepts that enable efficient automatic signaling: *closure of predicates, relay invariance*, and *predicate tagging*. The technique of *closure* of a predicate P is used to reduce the number of context switches for its evaluation. In the current systems, only the thread that is waiting for the predicate P can evaluate it. When the thread is signaled, it wakes up, acquires the lock to the monitor and then evaluates the predicate P. If the predicate P is false, it goes back to waiting. This results in an additional context switch. In our system, the thread that is in the monitor evaluates the condition for the waiting thread and wakes it only if the condition is true. Since the predicate P may use variables local to the thread waiting on it, ActiveMonitor derives a closure predicate P' of the predicate P, such that other threads can evaluate P'.

The idea of *relay invariance* is used to avoid *signalAll* calls in ActiveMonitor. We ensure that if there is any thread whose waiting condition is true, then there exists at least one thread whose waiting condition is true and is signaled by the system. With this invariance, the *signalAll* call is unnecessary in our automatic-signal mechanism. With relay invariance, the privilege to enter the monitor is transmitted from one thread to another thread whose condition has become true. This mechanism guarantees progress, and reduces the number of context switches by avoiding *signalAll* calls.

The idea of *predicate tagging* is used to accelerate the process of deciding which thread to signal. All the waiting conditions are analyzed and tags are assigned to every predicate according to its semantics. To decide which thread should be signaled, we identify tags that are most likely to be true after examining the current state of the monitor. Then we only evaluate the predicates with those tags.

We extend these concepts to task based executions by allowing conditions within asynchronous tasks. As defined in Defn. 1, each task has a boolean predicate P. This predicate captures the pre-condition for the task's execution. Before executing any task, the server thread must verify that this condition is true. If not, the task is not executable and the server does not execute it. The runtime handling of conditional synchronization for OR/AND operators is described in § 5.

4.2 Execution of Monitor Tasks

ActiveMonitor runtime library executes monitor tasks using the following rules.

▶ **Rule 1** (Mutex Invariant). *If some thread t is executing a task m of monitor M, then no other thread can execute any task m' of M concurrently.*

This rule maintains the mutual exclusion of critical sections of a monitor. We require two additional rules to guarantee execution of tasks in program order. Let $proc(t)$ denote the worker thread that submits the task t to a monitor. Let $sub(t)$ and $exe(t)$ respectively indicate the timestamps when t is submitted to the monitor, and when the server thread starts executing t.

▶ **Rule 2.** *For a pair of tasks s and t submitted to a monitor M, if $proc(s) = proc(t)$, then $sub(s) < sub(t) \Rightarrow exe(s) < exe(t)$.*

This rule ensures that a server (monitor thread) executes every worker's tasks in the program order of worker.

▶ **Rule 3.** *Let m_1, m_2 be two successive method invocations by a worker thread on two different monitors M_1 and M_2 in the user program, and let t_1, t_2 be their corresponding task submissions at runtime. Then, t_1 must be completed before t_2's submission.*

This rule enforces the constraint on a thread's successive invocations of methods on different monitor objects. Blocking method invocations in between these two calls are acceptable.

The notions of method *invocation* and *response* used to define linearizability [21] need a different interpretation under asynchronous executions. In short, *invocation* now corresponds to submission of the equivalent task to monitor thread, and *response* corresponds to this task's completion. Observe that the legal sequential history we get may not preserve the order of invocation of operations, but only the thread order. With this interpretation, we can easily validate the following result.

▶ **Lemma 2.** *Rules 1, 2 and 3 guarantee executions equivalent to lock-based executions.*

5 Compositionality: Multi-object Synchronization

Monitor tasks are compositional in nature. Suppose a monitor method declares n in the form of waituntil(P_i) S_i, where $1 \leq i \leq n$, to enforce that the set of statements S_i must be executed iff predicate P_i is true. To execute this method, ActiveMonitor generates n tasks such that each task t_i has a precondition P_i and a corresponding set of statements S_i. More importantly, with monitors allowed to be 'active' as threads, ActiveMonitor enables compositionality of blocking operations across different monitor objects. Consider two instances Q1 and Q2 of a blocking queue implementation, with dequeue method signature being deq(). As the queue is blocking, a call to deq() will block the calling thread if the queue is empty. Consider the problem of dequeueing from either of these instances, and storing the returned item into a variable x. If both queues are empty, then we should block until an item is available in either one. In ActiveMonitor, the code is simply one statement: x = Q1.deq() OR x = Q2.deq(). Solving this problem using the traditional mutex based blocking queue implementations is extremely difficult [19]. An *ad hoc* solution is to use a global lock and a lock-free/wait-free implementation of deq. But this solution does not scale because a global lock inhibits parallelism. Even with transactional memory [19] the problem is not easy to solve. To the best of our knowledge, no transactional memory implementation provides explicit wait/notify construct on individual thread-safe objects to release the CPU. An implementation [38] to allow waiting in transactional memory requires continuous loop based busy-waiting on conditions. Implementations such as [9] propose global lock based solutions for waiting and thus curb parallelism. Not only ActiveMonitor's asynchronous execution approach provides an elegant solution, but it also allows parallelism. Similarly, the AND operator allows conjunction of operations across multiple monitor objects, such that these operations can be performed in parallel.

5.1 Implementing AND & OR Operators in ActiveMonitor

For both of these operators, ActiveMonitor stipulates that the operands – monitor method calls – must be on different monitor objects. This is needed to guarantee program order under conditional synchronization across monitors. The pre-processor raises a parsing error if this constraint is not met. If the constraint is met, the pre-processor generates the equivalent task for each operand conjunct/disjunct clause, and stores them as a collection within a container object that is directly mapped to the operator. Note that if there are multiple statements with same operator usage, all of them are treated as independent, and a container object is generated for each of them. The operand calls are then replaced by the submission of tasks to the corresponding monitors.

The runtime library delegates the tasks to their respective target servers (monitor threads) for execution. It also observes all the preconditions of these tasks and ensures that they are executed whenever these conditions are met. For AND operator, the worker that called the operator is forced to wait for the completion of all the tasks. This is achieved by forcing the worker to evaluate the future reference returned by each task submission. Once all the futures have been evaluated, the result of the operator is stored in the designated storage if needed. For example, consider the statement: Q1.enq(a) AND Q2.enq(b); where Q1 and Q2 are two bounded-queues. Then the framework generates two tasks t1 and t2, and submits them to the server threads of Q1 and Q2. It then registers the returned future references with the worker thread that called the statement, and forces it to evaluate both the futures such that the worker remains blocked until both a and b are enqueued in Q1 and Q2 respectively.

For statements with OR operator, the container object that holds the tasks – that are equivalent to the constituent disjunct clauses of OR– also maintains an atomic flag called *taken*. This flag is initially set to false. To execute the composition statement, the runtime first parks the calling worker thread, and submits the tasks stored in the container object to their respective server (monitor). Recall that the *relay invariance* of our automatic signaling ensures that whenever the pre-condition of some task of the OR is met, its server thread is signaled. To guarantee that only one clause (equivalent task) of the OR statement is executed, the server thread performs a compare-and-swap (CAS) operation on the *taken* flag of the container object. If and only if the server's CAS operation succeeds, ie. the value of the flag was false and this server set it to true, the server proceeds to execute the task submitted to it. Since only one thread can succeed in atomically setting the flag, we are guaranteed that only one of the tasks will be executed. Every other server thread that executes the CAS and fails can discard its task for the OR statement.

6 Implementation

We now describe implementation details that make ActiveMonitor practical in terms of use with real world applications, as well as scalable and faster. Recall that unlike other delegation/combining implementations [35, 15, 10], threads do not perform busy-wait in ActiveMonitor. To enable conditional wait and yielding the CPU, our implementation uses a read/write lock for executing updates on each server thread. This ensures: (a) reads do not return stale values, and (b) servers/workers can release the CPU and go into waiting state whenever required as per runtime conditions. We employ a modified version of combining [15, 10] for executing critical section updates. When submitting a task to a monitor, a worker thread checks if the server of the monitor is in waiting state. If so, the worker acquires the lock – becomes the *combiner* – and executes a predefined number (five in our implementation) of tasks before releasing the lock. Observe that the actual acquisitions of the write-lock are mostly uncontended under this approach. Uncontended lock acquisitions are known to be relatively inexpensive, and thus threads does not incur significant performance penalty in doing so. For asynchronous tasks, we use a lightweight version of future objects that are shared between only one worker thread and the server. Only the server can update the state of these objects. Instead of using the default ones provided by the Java concurrent library [30], we create these objects using only a few volatile variables. Instead of using the default wait/notify mechanism provided by Java, we use the lower level API of park and unpark [30] for threads. Using the lower level API allows a more fine-grained control on execution of these threads.

6.1 Storage of Tasks: Single Consumer Optimal Bounded Queue

Although asynchronous executions generally benefit the application performance, a large number of asynchronous tasks in the system lead to degraded performance due to higher number of cache misses. To prevent this, ActiveMonitor maintains a bounded FIFO queue for each server in which the workers enqueue their tasks. Given that ActiveMonitor instantiates only one server thread (if any) per monitor object, this bounded-queue is a special case of the producer-consumer problem with only one consumer and multiple producers. Only the server consumes the items (tasks) from this queue, and all the workers produce the items. For this use-case, we developed an optimized algorithm for a thread-safe bounded FIFO queue that minimizes the synchronization costs for the consumer. The pseudocode of this algorithm can be found in the technical report at [1].

Our BoundedQueue is backed by a linked-list: the items are stored in the nodes of the linked-list. Only insertions in the queue require guarded execution under a lock to ensure correctness while multiple threads concurrently attempt to insert items. Only a single thread performs removal of items, and thus we do not require a lock to protect concurrent removals. However, maintaining the correct count of actual number of items in the queue is essential. This is done using the atomic integer count. We adopt a 'stealing' strategy in which the consumer locally caches the number of available items in a look-ahead manner and reads and updates the atomic integer count only when needed. Hence, the number of upadates to the atomic integer count is kept low, which in turn reduces the cache-coherence traffic, and

Table 1 Short description of problems evaluated.

Name	Short Desc.	CS Work [Type]	Details
PSSSP	Parallel single-source-shortest-path using Dijkstra's algorithm [7] using priority queue.	$\mathcal{O}(\log n)$ [Heavy]	(a) USA road network graphs (b) R-MAT Graphs [5]
BQ	Bounded FIFO queue of plain Java objects.	$\mathcal{O}(1)$ [Light]	Capacity varied from 4 to 64; number of enqueuers is equal to the number of dequeuers.
SLL	Linked-list of integers; entries are kept sorted in non-decreasing order.	$\mathcal{O}(n)$ [Heavy]	(a) Read-heavy: 90% reads, 9% insert, 1% delete (b) Write-heavy: 0% reads, 50% insert, 50% delete (c) Mixed: 70% reads, 20% insert, 10% delete
RR	Round-robin monitor access from [24].	$\mathcal{O}(1)$ [Light]	each thread accesses monitor in a predefined round-robin manner based on thread-id.

improves the throughput and scalability. Whenever there is no task (in its bounded-queue) for the server to execute, it is forced to go into wait. The server performs this wait outside the queue using a condition variable that it owns. The automatic signaling mechanism of the runtime library ensures that it is signaled and wakes up from the wait if a new executable task is enqueued in the queue.

For the single consumer multiple producer use-case, the throughput of our implementation is significantly higher than those of queue implementations from Java's util.concurrent package. The throughput comparison results on a saturation based micro-benchmark can be found in [1].

6.2 Monitor Thread Management

If we spawn a new thread for every monitor object, the performance of programs with relatively large number of monitors could suffer. ActiveMonitor allows the programmer to manually control this number, as well as itself controls the number of monitor threads based on the system hardware resources. The programmer can indicate an upper bound on the number of monitor threads when starting the application. The ActiveMonitor runtime library uses this limit in restricting the number of monitor threads spawned. If this limit is reached, no other monitor threads are created, and invocations of asynchronous methods on remaining monitors (that are not instantiated as threads) also follow the conventional synchronous (blocking) execution.

Irrespective of the user provided upper bound on server threads, the runtime library only instantiates a thread for a monitor if there is sufficient hardware available. The runtime library monitors the system environment information: CPU usage (for example from /proc/stat on Unix), and the size of wait-queues of monitor objects, to decide whether or not monitors should be executing as threads. If the CPU usage is high, our framework switches to traditional locking.

7 Evaluation

We implement monitor based solutions to multiple concurrency problems using ActiveMonitor, ReentrantLocks from JDK7, and combining [10] – that does not perform continuous busy-waits – by executing ActiveMonitor in only synchronous mode. We evaluate the performance of these implementations on light and heavy critical sections. Light critical sections (relatively small number of operations) do not involve much work within them, and favor traditional lock-based monitors as the overhead of maintaining additional information for delegated executions outweighs their benefits. On the other hand, heavy critical sections (large number of operations within CS) provide increased opportunity for exploiting asynchrony and parallelism. Table 1 presents a summary of problems used for our evaluation.

All the experiments are conducted on a 40-core Intel Xeon machine that consists of four sockets of Xeon E7-4850 10-core (20 hyper-threads), running at 2 GHz with 32 KB L1, 256 KB L2, and 24 MB LLC, respectively. Compilation and execution both are performed with Oracle Java 1.7 (64-bit VM). Across all results, we denote the implementations with the following notation: LK: implementation using Java's ReentrantLock, AM: ActiveMonitor with asynchronous executions, and AMS: ActiveMonitor running with only synchronous delegations.

For PSSSP problem, a thread-safe priority queue is used as an underlying data structure. ActiveMonitor solution of this problem uses the monitor-based implementation of an unbounded blocking priority queue from Java's concurrency package java.util.concurrent, and only modifies it to make the put method asynchronous. We evaluate the time taken to compute the shortest paths to all vertices from a randomly selected source vertex. We use five large sized directed graphs. Two of these graphs, FLA and NY, are USA road-network graphs of Florida, and New York obtained from [8], and the remaining three graphs: R16, R128, and R512 are generated using the GTGraph [2] generator suite. The three synthetic graphs – R16, R128, and R512 – have 5×10^4 vertices each, and 1.6×10^6, 1.28×10^7, and 5.12×10^7 edges respectively.

For all other problems we collect the throughput of operations over a 2 second period with varying number of workers. For BQ problem, the items in queue are randomly generated strings, with enqueue operation being asynchronous and dequeue being synchronous. For SLL problem, we pre-populated the data structure with 1000 entries to simulate steady state behavior. For all the operations, the operand values are chosen uniformly at random between 0 and 2000. This guarantees that on average, half of the operations are successful and the structure size does not grow too large. Insertions and deletions in the list are asynchronous and searches are synchronous. For RR, all accesses to the critical section are synchronous. BQ and RR problems require threads to perform conditional waiting. For these two problems, we also compare the performance of ActiveMonitor with that of Queue Delegation Locking [26], denoted by QD notation, by adding conditional waiting to QD. The purpose of this comparison is to establish that our approach of using automatic signaling with asynchronous executions can out-perform QD's approach of asynchronous delegation under lock-unavailability. In addition, we also compute throughput of performing OR implementations. For logical-or operations, we also tried to evaluate the performance of a transactional memory implementation [40] but this implementation resulted in runtime errors and could not execute the statements.

We perform multiple warm-up runs to negate just-in-time compilation related performance variations. In addition, all threads perform a fixed number of warm-up operations before starting the time measurements. For all the experiments, we collect runtimes for 7 runs, and report the mean value of 5 runs after discarding the highest and lowest values.

7.1 Results

Fig. 3 plots the throughput of the three PSSSP implementations in edges traversed per unit time format. Given that the three synthetic R-MAT [5] generated graphs are relatively dense in comparison to the road network graphs NY and FLA, the throughput values for all the implementations are higher for these graphs. AM outperforms both of LK and AMS. Specifically, on R512 graph – one with the highest density – AM is much faster than the other two. Given that the same implementation of priority queue is used as the underlying data structure for all three implementations, and the only difference is in terms of asynchronous inserts, these results validate our claim that AM approach is much more beneficial for heavy critical sections.

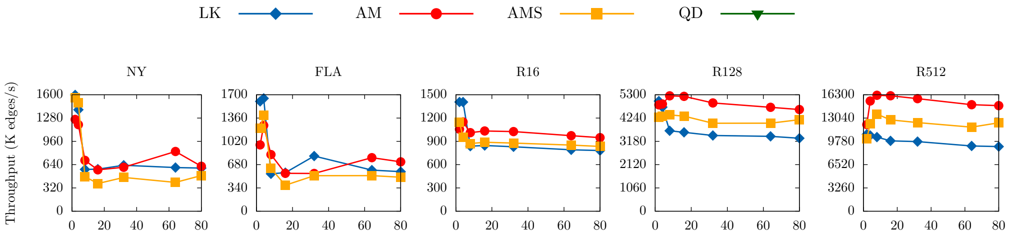


Figure 3 Throughput for PSSSP using priority queue (x-axis shows the number of threads)

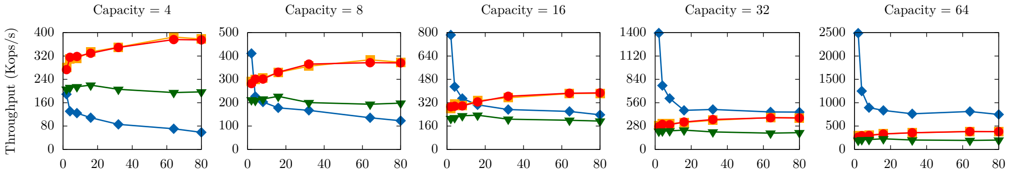


Figure 4 Throughput for Bounded FIFO Queue (x-axis shows the number of threads)

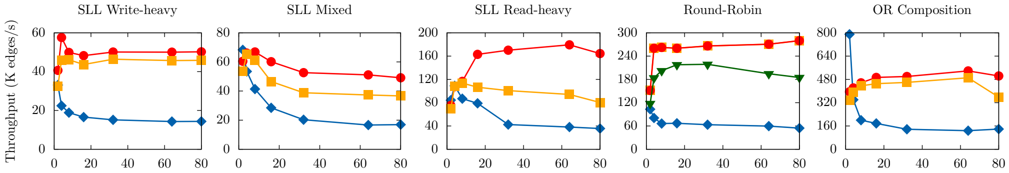


Figure 5 Throughput for SLL, RR, and OR problems (x-axis shows the number of threads).

Fig. 4 plots throughput of operations for different capacities of bounded queues for three implementation techniques. For smaller buffer sizes, in the range of 4 to 16 AM significantly outperforms LK implementation. This result highlights the benefits of asynchronous executions because LK is much slower in comparison to AM, as well as AMS due to high contention on locks. For larger capacities of 32 and 64, LK implementations perform better than AM because the availability of sufficient storage space allows worker threads to repeatedly acquire critical sections without being blocked out, and LK benefits from Java's policy of non-fairness in lock acquisitions. In contrast, AM and AMS provide *almost* 'fair' executions for workers. However, in doing so, they end up performing more work in these cases where blocking due to unavailability of space occurs rarely. In the technical report version [1] of this paper, we analyze the performance benefit of asynchronous delegation by dropping the program order constraint and conducting the same experiment. In the new setting, AM performance further improves, and outperforms LK even on capacity of 32 (see Fig. 7 in [1]). These results highlight that when asynchronous executions are allowed to be out of program order, the overall throughput of the program can improve significantly.

Fig. 5 shows the operations throughput for the SLL and RR, and OR composition problems. In all the runs on these problems, (AM) significantly outperforms the read-write reentrant lock based monitor (LK), as well as delegation technique of AMS. Note that RR problem does not involve any asynchronous operation, and thus AM and AMS runs are exactly the same. Given that the critical section involved in SLL problem is heavy, the performance gap highlights the benefits of asynchronous monitors for such cases. Surprisingly, AM (as well as AMS) is ∼3–4× faster than LK on RR problem too. This is because the RR problem setup simulates a critical section that is similar to BQ problem with capacity one. Hence, LK implementation spends a lot of its execution time in waiting for lock acquisitions, whereas AM and AMS benefit from lower contention.

On all the problems with conditional waits, AM significantly outperforms QD in terms of throughput. Hence, extending QD to incorporate conditional waiting is not sufficient to match our approach. Our techniques for efficient conditional synchronization with automatic signaling provide significant benefits in comparison to QD.

8 Related Work

Our idea of having monitor objects execute as independent threads is influenced by Hoare's proposed communicating sequential processes (CSP) [23] mechanism in which all objects are *active*, of long ago. However, CSP does not have the notion of shared memory, and every object is a process. In contrast, our focus is solely on shared memory parallel programs on multi-core machines.

We use *futures* [12, 30] to realize the idea of non-blocking/asynchronous executions. Kogan et al. [27] explore a similar approach in making use of *futures* for non-blocking executions. However, we explore changes to the general paradigm of monitors, whereas [27] only focuses on three data structures: stacks, queues, and linked-lists, none of them requiring conditional waiting. In addition, [27] uses data structure specific local elimination/combining, and allows read/fetch operations on these data structures to be asynchronous whereas we do not – our assumption being that in almost all the cases, a programmer needs the result of read/fetch immediately so that she can use it in the subsequent program logic. Hence, our approach spans a more generic level of monitors, and does not rely on knowledge of internal functionality of critical section it protects. Some theoretical results that establish the bounds on improvements in cache locality by the use of futures have been established in [17]. These results are not directly related to monitor based executions, but lead the direction in terms of use of futures for improving the performance of multi-threaded programs.

Existing implementations of the combining technique [35, 10, 15] perform busy waits for task completions and do not yield the CPU; additionally they also do not provide any mechanisms for conditional waits – these issues together make them more or less impractical for use in real world applications. Remote Core Locking (RCL) [31] addresses such issues by allowing conditional waits, and uses a dedicated core for executing critical section, but does not incorporate asynchronous executions. Recently, works such as [36, 4] have performed extensive experimental analysis in identifying the performance gains/losses with asynchronous message-passing like executions over synchronous shared memory ones. [36] provides various insights for effective implementations that perform well using hardware message passing support on shared memory machines. This work minimizes the remote-memory-references (RMRs) during executions, and quantifies the performance gains for asynchronous executions, but assumes that the method data fits in a single cache-line. In addition, it does not consider the conditional wait based monitor implementations. Similarly, [4] studies the pros and cons of message passing based executions on performance of shared memory parallel programs. This work highlights that different approaches perform best under different circumstances, and that the communication overhead of message passing can often outweigh its benefits, and discusses ways in which this balance may shift in the future. Queue Delegation Locking (QDL) [26], uses the approach of combining to provide a locking library implementation in C++. However, QDL does not provide a mechanism for synchronization between threads, and waiting, based on conditions.

Transactional memory [18, 37] is a well-known research effort that proposes modified syntax for ease of writing multi-threaded programs. However, constructs for conditional waiting under transactional memory are limited [38, 32, 9]. Hence, writing many conditional synchronization based multi-threaded programs is rather difficult. Also, unlike transactional memory, our approach merely transfers the responsibility of data manipulation to monitor threads and does not require any complicated rollback mechanism for resolving conflicting updates on the shared data. x10 [6] programming language focuses on providing features that have an overlap with both transactional memory and our work. However, there are significant differences in the support and usage of these constructs. The support for conditional waiting is present syntactically, but as stated in [6] is deprecated for runtime execution.

Lock-free algorithmic techniques using atomic hardware instructions such as *compare-and-swap* have gained momentum for implementing scalable thread-safe data structures [13, 33, 16, 11, 20, 28, 29, 39, 34]. In addition, [14, 25] have explored alternate implementation techniques that combine/eliminate complementary operations for increasing parallelism in data structures. However, the difficulty involved in designing lock-free/wait-free algorithms, and operation eliminating data structures is well known. At present, it is not clear how lock-free techniques can be used to implement critical sections that involve many operations spanning across multiple shared objects. The absence of any wait-notify mechanism in lock-free techniques is another hurdle for their use in many real world programs.

9 Discussion & Conclusion

Despite providing programming ease and performance benefits, our framework's current implementation has some limitations. We discuss them below.

Thread Dependent Variables and Functions: In our current implementation, thread dependent variables and functions within a monitor method cannot be used directly in the Runnable or Callable object that is used in task generation by our approach. This is because the tasks are executed by the monitor thread and not by the worker thread. For example, suppose there is a monitor method that invokes Thread.currentThread(), if we directly add this statement to the generated Runnable object (in the task), then this method's invocation at runtime will return the reference to the monitor thread when it is executed. However, it is obvious that the intent of this call inside the monitor method was to refer to the worker thread. To handle this situation, currently, we require the programmer to perform reference copy and storage and storage in thread-local variables. For read operations of thread dependent variables and functions, the worker thread would need to evaluate them outside the monitor, and store the result with final variables. These final variables can be accessed by the runnable and callable objects. An additional constraint/limitation applies for the case of write operation on thread dependent variables. For write operations, if the monitor method is non-blocking then the results can be stored as intermediate data. The worker thread then writes these results back to its local variable after the task is executed.

Concluding Remarks: We have shown that our proposed scheme of asynchronous executions in monitors provides significant improvement over traditional lock-based monitors. At present, writing parallel programs that provide high throughput and scalability is an arduous task for most programmers. The main challenge is a lack of simple programming language constructs that guarantee thread-safety while exploiting parallelism of executions and availability of hardware in a seamless and portable manner. Our proposed design of asynchronous monitors is a step in the direction of providing such constructs. The current version of our implementation consumes some additional processing resources. We believe, however, that with further research efforts in this direction our proposed technique can lead to significant improvements in programmability as well as performance of shared memory parallel programs.

References

1 ActiveMonitor: Technical Report Version. `http://pdsl.ece.utexas.edu/TechReports/2016/opodis_tr.pdf`.

2 David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

3 Peter A. Buhr, Michel Fortier, and Michael H Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

4 Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *OPODIS*, pages 83–97, 2013.

5 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

6 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

7 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959. `doi:10.1007/BF01386390`.

8 9th DIMACS Implementation Challenge – Shortest Paths. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

9 P. Dudnik and M. Swift. Condition variables and transactional memory: Problem or opportunity? In *The 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

10 Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. *ACM SIGPLAN Notices*, 47(8):257–266, 2012.

11 Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC'04, pages 80–87, 2004.

12 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

13 Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

14 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, WilliamN III Scherer, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, pages 3–16. Springer Berlin Heidelberg, 2006.

15 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *SPAA*, pages 355–364, 2010.

16 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'04, pages 206–215, 2004.

17 Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14, Orlando, FL, USA, February 15-19, 2014*, pages 155–166, 2014.

18 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA'93, pages 289–300, 1993.

19 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.

20 Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC'88, pages 276–290, 1988.

21 Maurice P Herlihy and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

22 C A R Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

23 C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

24 Wei-Lun Hung and Vijay K. Garg. AutoSynch: An Automatic-signal Monitor Based on Predicate Tagging. In *PLDI*, pages 253–262, 2013.

25 Joseph Izraelevitz and Michael L. Scott. Brief announcement: a generic construction for nonblocking dual containers. In *ACM Symposium on Principles of Distributed Computing, PODC'14*, pages 53–55, 2014.

26 David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par*, 2014.

27 Alex Kogan and Maurice Herlihy. The future (s) of shared data structures. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 30–39. ACM, 2014.

28 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP'11, pages 223–234, 2011.

29 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, pages 141–150, 2012.

30 Doug Lea. The Java.Util.Concurrent Synchronizer Framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

31 Jean-Pierre Lozi et al. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX Annual Technical Conference*, pages 65–76, 2012.

32 V. Luchangco and V. J. Marathe. Revisiting condition variables and transactions. In *The 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

33 Maged M Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

34 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328. ACM, 2014.

35 Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on*

Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA'99). World Scientific, 1999.

36 Darko Petrovic, Thomas Ropars, and André Schiper. Leveraging hardware message passing for efficient thread synchronization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14, Orlando, FL, USA, February 15-19, 2014*, pages 143–154, 2014.

37 Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 204–213, 1995.

38 A. Skyrme and N. Rodriguez. From locks to transactional memory: Lessons learned from porting a real-world application. In *The 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.

39 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'12, pages 309–310, 2012.

40 TMWare – TMJava. `http://tmware.org/`.

Communicating with Beeps*

Artur Czumaj[1] and Peter Davies[2]

1 **Department of Computer Science, Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, Warwick, UK**
A.Czumaj@warwick.ac.uk

2 **Department of Computer Science, Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, Warwick, UK**
P.W.Davies@warwick.ac.uk

Abstract

The *beep model* is a very weak communications model in which devices in a network can communicate only via beeps and silence. As a result of its weak assumptions, it has broad applicability to many different implementations of communications networks. This comes at the cost of a restrictive environment for algorithm design.

Despite being only recently introduced, the beep model has received considerable attention, in part due to its relationship with other communication models such as that of ad-hoc radio networks. However, there has been no definitive published result for several fundamental tasks in the model. We aim to rectify this with our paper.

We present algorithms for the tasks of broadcast, gossiping, and multi-broadcast, and also, as intermediary results, means of depth-first search and diameter estimation. Our $O(D + \log M)$-time algorithm for broadcasting is a simple formalization of a concept known as beep waves, and is asymptotically optimal. We give an $O(n \log L)$-time depth-first search procedure, and show how this can be used as the basis for an $O(n \log LM)$-time gossiping algorithm. Finally, we approach the more general problem of multi-broadcast. We differentiate between two variants of this problem: one where nodes must know the origin of all source messages, and another where this information is not required. In the first instance we achieve an algorithm running in time $O(k \log \frac{LM}{k} + D \log L)$, and in the second an $O(k \log \frac{M}{k} + D \log L)$-time algorithm (or $O(M + D \log L)$ when $M \leq k$). We then give corresponding lower bounds: $\Omega(k \log \frac{LM}{k} + D)$ in the case where nodes must know message origins, and $\Omega(k \log \frac{M}{k} + D)$ and $\Omega(M + D)$ in the other case, for $M > k$ and $M \leq k$ respectively. These lower bounds demonstrate that our algorithms are optimal except for the $D \log L$ additive term. In these running-time expressions, n represents network size, D network diameter, L range of node labels, M range of source messages, and k number of sources.

Our algorithms are all explicit, deterministic, and practical, and give efficient means of communication while making arguably the minimum possible assumptions about the network.

1998 ACM Subject Classification C.2.1 Distributed Networks

Keywords and phrases Beep model, Communication networks, Broadcasting, Gossiping, Leader election

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.30

[*] Research partially supported by the Centre for Discrete Mathematics and its Applications (DIMAP).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *beep model*, introduced recently by Cornejo and Kuhn [3], is a very weak network communications model in which information can be passed only in the form of a beep or a lack thereof. The model is related to the ad-hoc radio network model, and has been used as a surrogate model in results concerning radio networks with collision detection. As well as attracting study from this angle, the beep model is interesting in its own right because of its generality, simplicity, and wide range of areas where it could be applied.

1.1 Model

The network is modeled as an undirected connected graph $G = (V, E)$, where vertices in the graph represent devices in the network, and edges represent direct reachability. Time is divided into discrete steps, with a synchronized global clock (though, as in [4], we can extend to the case where only a subset of nodes wake up at time 0 and others must be woken by receiving beeps). In each time-step every node decides whether to *beep* or to *listen*. Nodes which choose to listen in a particular time-step hear a beep if at least one of their neighbors chose to beep, and they cannot distinguish between one neighbor beeping or many. We will assume that nodes have unique labels (IDs), which is essential (at least when considering deterministic algorithms) in order to break symmetry.

We will use the following parameters in analysis of our algorithms:

- n will denote network size, i.e., $|V|$.
- D will denote network diameter, the largest distance between any pair of nodes.
- L will be the range of node labels, i.e., labels will be strings of no more than $\log L$ bits.
- M will be the range of messages, i.e., messages will be strings of no more than $\log M$ bits.
- k will be the number of source nodes when considering the multi-broadcast task.

We do not, however, assume that nodes have any prior knowledge of these parameters, nor any other knowledge about the network.

1.2 Related Work

There has been a large amount of research focusing on fundamental communication problems in distributed computing, see e.g., [9] and the references therein. The beep model was introduced by Cornejo and Kuhn [3], who used it to design an algorithm for interval coloring. This task is a variant of vertex coloring used in resource allocation problems, and is, in a sense, tailored to the model. In another recent work, Afek et al. [1] presented an algorithm for finding a maximal independent set in the beep model. The beep model is strictly weaker than the model of radio networks with collision detection (see, e.g., [9]), though the aforementioned two results did not approach it from this angle, and so algorithmic results in the former also apply in the latter. This relationship was exploited by Ghaffari and Haeupler [6] to give almost optimal $O((D + \log n \log\log n) \cdot \min(\log\log n, \log \frac{n}{D}))$-time randomized algorithm for leader election in radio networks with collision detection. Ghaffari and Haeupler [6] introduces the method of "beep waves" to transmit bit strings, a method which is also employed here for the purpose of broadcast. Ghaffari et al. [5] give a randomized broadcast algorithm in radio networks with collision detection which employs beeping techniques, but, unlike the algorithm of [6], does not entirely translate over to the beep model. A deterministic leader election algorithm in the beep model was given by Förster et al. [4], taking $O(D \log L)$ time. While a simple binary search approach, like that used in [2] for radio networks, gives the same running time, the method of [4] has the benefit of not requiring prior knowledge of parameters D and L, an advantage which we make use of in our results. In another related work, Gilbert and Newport [7] studied the quantity of computational resources needed to solve specific problems in the beep model.

Concurrently with this paper, Hounkanli and Pelc [8] give a $O(D + \log M)$ time broadcasting algorithm and an $O(n^2 \log M + nD \log L)$-time gossiping algorithm in a slightly different model where nodes know network parameters n, L, M but wake-up at arbitrary different time-steps, rather than simultaneously. To our knowledge there have been no earlier published results for broadcast, gossiping, and multi-broadcast in the model we study.

1.3 Our Results

In this paper, we present the following results:

- An optimal $O(D+\log M)$-time algorithm for broadcasting a $\log M$ bit message, developing and formalizing the "beep waves" method of [6].
- An $O(n \log L)$-time procedure for performing depth-first search.
- An $O(n \log LM)$-time gossiping algorithm based on depth-first search.
- An $O(D)$-time procedure for estimating diameter.
- An $O(k \log \frac{LM}{k} + D \log L)$-time algorithm for multi-broadcast with provenance (where every node must learn all (source ID, source message) pairs).
- A corresponding $\Omega(k \log \frac{LM}{k} + D)$ lower bound.
- An algorithm for multi-broadcast without provenance (where every node must learn all unique source messages) taking $O(k \log \frac{M}{k} + D \log L)$ time when $M > k$ and $O(M + D \log L)$ time when $M \leq k$.
- A corresponding lower bound of $\Omega(k \log \frac{M}{k} + D)$ when $M > k$ and $\Omega(M + D)$ when $M \leq k$.
- These multi-broadcasting algorithms imply $O(n \log \frac{LM}{n} + D \log L)$ and $(n \log \frac{M}{n} + D \log L)$-time gossiping algorithms with and without provenance respectively.

The multi-broadcasting algorithms are our most significant results. The first outperforms the DFS-based gossiping algorithm despite being designed for a more general problem. Furthermore, perhaps surprisingly, the second is faster even than the $k \log M$ time-steps required for a node to directly transmit or hear the source messages, which might intuitively have appeared to be a lower bound for the problem.

2 Broadcasting

Broadcasting is perhaps the most fundamental task in distributed communication models. It assumes that one designated source node has a message (which we will assume to be an integer in the range $[0, M - 1]$) that must be known by all nodes in the network. We achieve optimal an $O(D + \log M)$-time algorithm for broadcasting based on the idea of "beep waves."

2.1 Beep Waves

Beep waves were first introduced by Ghaffari and Haeupler [6] as a means of transmitting information in the beep model. Variations of the technique are useful for different circumstances, and here we give a simple formalization tailored to the task of broadcasting from a single source.

The idea is the following: every three time-steps, starting at zero, the source transmits a bit of its message, that is it beeps to represent a **1** or remains silent to represent a **0**. We can encode the message so that it is obvious when the beginning and end are, for example by duplicating every bit of the message and then placing **10** at the beginning and end. We will denote this coding method C, and note that for any message m, $|C(m)| \leq 2|m| + 4$. When we refer to the size of the message, we mean its length in bits, i.e. $|m| \leq \log M$. It is easy to see that we can decode to find the original message(s), even if there are several, separated by any number of **0**s. This will become necessary within algorithms for more complex tasks which involve several successive broadcasts.

All non-source nodes, upon hearing a beep in some time-step i, then relay the beep themselves in time-step $i + 1$, unless they themselves beeped in time-step $i - 1$.

Algorithm 1 BEEP-WAVE$(s, m(s))$ at source s

```
for $i = 1$ to $|C(m(s))|$ do
    if bit $C(m(s))_i$ is 1 then
        $s$ beeps in time-step $3i$
    end if
end for
```

Algorithm 2 BEEP-WAVE$(s, m(s))$ at non-source u

```
while end of message not heard do
    if $u$ hears a beep in time-step $i$ and did not itself beep in time-step $i - 1$ then
        $u$ beeps in time-step $i + 1$
        bit $m(u)_{\lfloor i/3 \rfloor} \leftarrow 1$
    end if
end while
output $C^{-1}(m(u))$
```

▶ **Lemma 1.** BEEP-WAVE$(s, m(s))$ *correctly performs broadcast in time* $O(D + |m(s)|) = O(D + \log M)$.

Proof. Partition all nodes into layers depending on their distance from the source s, i.e. layer $L_i = \{v \in V : dist(v, s) = i\}$. Every beep emitted by the source is propagated one layer per time-step, reaching all nodes in layer i after i time-steps. Nodes in layer i only ever relay beeps from layer $i - 1$, because the only times layer $i + 1$ beeps are directly after layer i does. This can be seen by an inductive argument.

Therefore, a node in layer i receives a beep exactly i steps after the source transmits one, and so can decode its received bit-string to recover the source's message. ◀

To use our broadcast algorithm, we must have a designated source node, and we must also have a good estimate of D if we want to know how much time to allow for completion. We do not require nodes to have access to a synchronized global clock when performing BEEP-WAVE; however, non-source nodes must know that they should be behaving in a "beep-forwarding" fashion. If we wish to use broadcast as part of larger algorithms, then we must take care to ensure that each node has the correct behavior during the time period involved.

2.2 Lower Bound for Broadcasting

It is straightforward to demonstrate that the $O(D + \log M)$ running time of BEEP-WAVE is asymptotically optimal:

▶ **Lemma 2.** *Any algorithm for broadcasting a message* $m(s) \in [0, M-1]$ *must take at least* $c(D + \log M)$ *time-steps for some constant* c*.*

Proof. The message can be any of M different bit-strings, and so requires $\log M$ bits to specify. Let u be the furthest non-source node from the source s. Since u can only receive at most one bit of information per time-step (a beep or silence), $\log M$ time-steps must be required for it to know the message.

Information can only be propagated through the network at most one adjacency layer per time-step, since non-adjacent nodes have no means of communication. Therefore at least $dist(u, s)$ time-steps are required for any information to reach u. Since $dist(u, s) \geq \frac{D}{2}$, the total number of time-steps required to inform u of the message is at least $\max(\frac{D}{2}, \log M) \geq \frac{1}{2}(\frac{D}{2} + \log M) \geq \frac{1}{4}(D + \log M)$ ◀

3 Leader Election

If we wish to use broadcast as part of a more complex algorithm, we must be sure that we have a single source who wishes to send a message. To ensure this, we can perform the task of leader election.

Leader election enables all nodes to agree on the ID of one particular node. In our applications, we will always choose the node with the highest ID in the entire network. More generally, though, leader election can be used on any subset of nodes, whenever each holds some integer value, to find the participating node with the highest (or lowest) such value. The values need not even be unique, since if multiple nodes hold the target value, we can pick out one by performing leader election again on their IDs. Leader election, particularly when used in this way, is sometimes also referred to as Find Max.

We wish to be able to perform leader election in $O(D \log L)$ time. We note that there is a straightforward way to do this: we can perform a binary search for the highest ID, iterating through the bits of the IDs and having all nodes who are still "in the running" for leader, and who have a 1 in the current position, broadcast. While we cannot use our previous broadcast procedure with multiple sources, since these nodes need only transmit a single bit we can still use beep-waves to ensure that the network hears *something*. This is sufficient for all nodes to determine whether any have a 1 in the current position. A similar method to this was used to perform leader election in radio networks in [2].

However, there is a problem with this approach: we would need a common linear upper bound on D and a polynomial upper bound on L to correctly perform it. Since we do not assume this knowledge, we instead make use of a result of Förster et al. [4] (paraphrased):

▶ **Theorem 3.** *There is an algorithm* ElectLeader *which performs leader election in time* $O(D \log L)$ *without prior knowledge of* D *or* L*.*

Upon completion, all nodes have knowledge of the highest ID, and can therefore use this as L in future operations. Förster et al. extend their algorithm to function when only some subset of nodes wake up at time 0, removing the assumption of synchronous wake-up. Since we employ it as a subroutine at the beginning of all our forthcoming algorithms, they can also forgo this assumption.

4 Network Traversal

We may wish to perform operations which require an organized exploration of the entire network. For this purpose, we give a procedure for depth-first search, and an application to the task of gossiping.

4.1 Depth-First Search

Depth first search is performed here in fundamentally the usual way. There is a network-wide "token," i.e., only one node is the 'active" node at any one time. This node checks for unexplored neighbors, passes the token to one if any exist, or sends it back to its parent if not. Here we also wish to pass round a counter, incremented upon reaching each new unexplored node, so that nodes know the order in which they were explored.

To detect unexplored neighbors we use a process much like the binary-search method mentioned for leader election, in which nodes iteratively agree on each bit of an ID. Here, though, we do not need to broadcast to the whole network between every step, since the nodes involved are all adjacent to the current active node. To organize this process we need predetermined constant-size control messages; this can easily be achieved by using any sensible system of code-words.

To apply Algorithm 4 we must first have a designated leader node. This leader is the parameter v taken as input in our description of the algorithm.

Algorithm 3 DEPTH-FIRST SEARCH$(v, x, count)$

```
$number(x) \leftarrow count$
loop
    $x$ transmits "child-acknowledge" message
    unvisited neighbors beep
    if $x$ received no beep then
        break loop
    end if
    $x$ transmits "child-search" message
    for $i = 1$ to $\log L$ do
        unvisited nodes still in running for highest ID beep if $i^{th}$ bit of ID is 1
        $x$ transmits "acknowledge 1" message if it heard a beep, "acknowledge 0" otherwise
        if "acknowledge 1" received, nodes with $i^{th}$ bit 0 drop out
    end for
    $y \leftarrow$ ID highest unvisited neighbor
    $x$ transmits the message (y,$count + 1$)
    $count \leftarrow$ DEPTH-FIRST SEARCH$(v, y, count + 1)$
end loop
if $v = x$ then
    terminate procedure
else
    return $count$ (by transmitting back to parent)
end if
```

▶ **Lemma 4.** DEPTH-FIRST SEARCH$(v, v, 1)$ *correctly performs depth-first search within* $O(n \log L)$ *time-steps.*

Proof. Each round of the "child-search" loop ensures that the current token node and its unexplored children all agree on a bit of the target ID, and since only one node has the token there will be no interference from the rest of the network. Thus, after $\log L$ such rounds, the ID of the next node to explore is agreed upon. This process must be performed n times to explore the entire graph, taking $n \log L$ total time-steps. We must also consider the cost of passing *count*, which is $O(\log n)$ time-steps each time the token moves, and so $O(n \log n)$ in total. Since $L \geq n$, total running time is $O(n \log L)$. ◀

4.2 DFS-Based Gossiping

An application of our depth-first search algorithm is to the task of gossiping. The premise of gossiping is that every node has a message which must become known to the entire network. We can achieve this in $O(n \log LM)$ time by first electing a leader, then performing depth-first search, and finally having each node broadcast its message in the order in which it was explored by the DFS.

This last broadcast stage is not quite as straightforward as it may seem, since in general n different consecutive broadcasts would take $O(n(D + \log M))$ time, exceeding our desired $O(n \log LM)$ running time. However, since we can encode messages so that it is obvious when they start and end (without affecting asymptotic size), and we also know that transmissions during Beep-Wave move exactly one distance layer per time-step and never move backwards, we can pipeline the broadcasts. That is, once a node's message, in its entirety, has been heard by the next node in the ordering, that next node can immediately begin its own broadcast without waiting for the previous message to reach the entire network. The waves of beeps will not interfere with each other, since the start of the new message cannot reach any node quicker than the end of the old message, and the behavior of all other nodes does not change, so they do not need to know the precise time-step when the new node starts broadcasting.

Algorithm 4 Gossip$(m(V))$

```
$v \leftarrow$ ElectLeader
perform Depth-First Search$(v, v, 1)$
for $i = 1$ to $n$ do
    let $u$ be such that $number(u) = i$
    Beep-Wave$(u, m(u))$
end for
```

▶ **Theorem 5.** Gossip *correctly performs gossiping within* $O(n \log LM)$ *time-steps.*

Proof. Performing ElectLeader and Depth-First Search takes $O(n \log L)$ time in total. Upon completion, each node knows its ordering in the DFS tree. Nodes then broadcast in order, and a node can begin broadcasting immediately after hearing the end of the message from its predecessor. The total time taken for all n broadcasts is then $O(\sum_{i=1}^{n}(\log M + dist(i, i+1)))$. Since this sum of distances is no greater than the distance traveled when traversing the DFS tree, this expression is $O(n \log M)$. Therefore total running time is $O(n \log L + n \log M) = O(n \log LM)$. ◀

5 Auxiliary Procedures

We next define some procedures for useful auxiliary tasks, which we will need for our multi-broadcast algorithms, but are also general enough to be useful elsewhere. Specifically, we give protocols for:

- Diameter estimation, i.e. allowing all nodes to calculate a common linear upper bound on D.
- Message collection, where a designated leader node receives the logical **OR**-superimposition of bit-strings from several source nodes.

- Message length determination, i.e. providing all nodes with the size of the largest of a set of bit-strings from source nodes.

5.1 Diameter Estimation

Our model assumes that nodes do not have access to any of the network parameters. In algorithms for complex tasks, we generally wish to start with a leader election phase, and this provides all nodes with knowledge of L. However, if we also wish to know the value of D, we must perform an extra task for this purpose.

Our diameter estimation procedure (Algorithm 5) works as follows: we take as input a leader node to co-ordinate the process. An initial beep from the leader propagates through the network. Having received this beep, nodes beep to acknowledge their existence back to the leader; a modularity restriction on when nodes can transmit ensures that these beeps only travel backwards through the layers. While the initial beep from the leader is still reaching further nodes, acknowledgment beeps will continue to return through the network every three time-steps. Once all nodes have been reached, this pattern will cease, and the leader will know that the diameter of the graph is no greater then the current time-step value. All of the other nodes have also ceased transmission, and so an application of Beep-Wave can safely be used to broadcast the diameter estimate.

We split the algorithm into two parts, one performed by the leader, and one performed by all non-leader nodes, since their behavior is quite different.

Algorithm 5 EstimateDiameter(v) at leader v

```
$v$ beeps in time-step 1
let $i$ be the first time-step (greater than 2) in which $v$ has not received a beep for 3 previous time-steps
$\tilde{D} \leftarrow i$
perform Beep-Wave($v, \tilde{D}$)
output $\tilde{D}$
```

Algorithm 6 EstimateDiameter(v) at non-leader u

```
let $j$ be the first time-step in which $u$ receives a beep
$u$ beeps in the next time-step which is equivalent to $(-j)$ mod 3
while $u$ has heard a beep in the last 3 time-steps do
    any beep $u$ hears in a time-step equivalent to $(2 - j)$ mod 3, it relays in the next time-step
end while
$\tilde{D} \leftarrow$ Beep-Wave($v, \tilde{D}$)
output $\tilde{D}$
```

▶ **Lemma 6.** EstimateDiameter(v) *correctly broadcasts an estimate* $\tilde{D}$ *satisfying* $D \leq \tilde{D} \leq 2D + 7$*, and terminates within* $O(D)$ *time-steps.*

Proof. Let D' be the distance from the leader to the furthest node. Then, $D \geq D' \geq D/2$. The leader emits a beep in time-step 1 which travels to this furthest node by time-step $D' + 1$. After at most 3 more time-steps, the node transmits its acknowledgment beep, which travels back to the leader in a further D' time-steps. After another 3 time-steps, the leader knows that it has received the final acknowledgment beep, and takes the current time-step i as its diameter estimate. Since $2D' \leq i \leq 2D' + 7$, the estimate $\tilde{D}$ which is broadcast to the network satisfies $D \leq \tilde{D} \leq 2D + 7$. Running time of the estimation phase is no more than $2D + 7$ time-steps, and the final broadcast takes $O(D + \log D) = O(D)$ time. ◀

Since we are only interested in asymptotic behavior, we will assume, for ease of notation, that having performed EstimateDiameter as part of a more complex algorithm we can then make use of the exact value of D.

5.2 Message Collection

We next introduce a sub-procedure (Algorithm 7) which will allow the leader to collect messages $m(S)$ from a set of sources S, receiving an **OR**-superimposition of all the messages. This works similarly to the usual beep-waves procedure, except that nodes use their distance from the leader (inferred by the time taken to receive the initial Beep-Wave$(v, \mathbf{1})$) to ensure that the waves only travel towards the source, and all messages arrive at the same time. We must have an input parameter p giving an upper bound on the length of messages, so that nodes know when the procedure is finished, and we assume that we have already performed EstimateDiameter and so can make use of D.

Algorithm 7 CollectMessages$(v, S, m(S), p)$ at source $s \in S \setminus \{v\}$

```
perform Beep-Wave($v, \mathbf{1}$)
for $i = 1$ to $p$ do
    if bit $m(s)_i$ is $\mathbf{1}$ then
        $s$ beeps in time-step $3i + D - dist(s)$
    end if
end for
```

Algorithm 8 CollectMessages$(v, S, m(S), p)$ at $u \notin S \setminus \{v\}$

```
perform Beep-Wave($v, \mathbf{1}$)
for $j = 0$ to $3p + D$ do
    if $u$ hears a beep in time-step $j$ such that $j \equiv 2 + D - dist(u) \bmod 3$ then
        $u$ beeps in time-step $j + 1$
        if $u = v$ then
            bit $m(u)_{\lfloor j/3 \rfloor} \leftarrow 1$
        end if
    end if
end for
output $m(v)$
```

We note that even if leader v is itself a source, it should perform the non-source behavior. Since it already knows its own message, it can superimpose it with the string it receives manually upon termination of the procedure.

▶ **Lemma 7.** CollectMessages$(v, S, m(S), p)$ *correctly informs v of the **OR**-superimposition of $m(S)$ within $O(D + p)$ time-steps.*

Proof. The modularity restriction on relaying beeps ensures that beep-waves only travel towards the v, and the starting times for sources ensure that bits in the same position arrive simultaneously. Thus, v hears a **1** in a position iff one of the sources messages contained a **1** in the same position. After $D + 3p$ time-steps it must then have received the **OR**-superimposition of $m(S)$. ◀

5.3 Message Length Determination

One issue with using CollectMessages is the necessity of prior knowledge of a common upper bound on message size. We give a simple method of obtaining this bound (Algorithm 9).

We perform CollectMessages using strings which are as long as the messages we actually want to collect, but consist of entirely **1**s. The superimposition of these strings is a **1**-string of equal length to the longest message. Since the leader will be able to tell that this string has ended when it hears the substring **10**, the procedure can be terminated even without an upper bound for the CollectMessages call.

Algorithm 9 GetMessageLength$(v, S, m(S))$

```
perform $p \leftarrow$ CollectMessages$(v, S, \mathbf{1}^{m(S)}, \infty)$, terminating when $v$ hears the substring 10
perform Beep-Wave$(v, |p|)$
output $|p|$
```

▶ **Lemma 8.** GetMessageLength$(v, S, m(S))$ *correctly informs all nodes of* $p = \max_{s\in S} |m(s)|$ *within* $O(D + p)$ *time-steps.*

Proof. CollectMessages will terminate after $D + 3p$ steps, since v will hear the final **1** and then a **0**. All other nodes will be inactive and so Beep-Wave$(v, |p|)$ will successfully inform the network of p (nodes will be aware that the CollectMessages phase is over and so perform Beep-Wave correctly, since they either heard a string of contiguous **1**s and then a **0** during CollectMessages, or silence for more than D time-steps).

Running time is $O(D + p)$ for CollectMessages and $O(D + \log p)$ for Beep-Wave, giving $O(D + p)$ total. ◀

6 Multi-Broadcast

We are now ready to approach the most general of the communication tasks we will consider, that of multi-broadcast. As in gossiping, multiple source nodes have messages which must become known to the entire network. However, rather than all nodes being sources, only those belonging to some unknown subset are. We denote the number of sources as k, but this value is not known to the network.

There are two slightly different variants of the problem we consider: *multi-broadcast with provenance*, where the network must become aware of all (source ID, source message) pairs, and *multi-broadcast without provenance*, where the IDs need not be known. Since we do not assume that messages are unique, we also allow in the case without provenance that only one copy of each distinct message must be output. That is, nodes need not be aware of how many sources held each message.

6.1 Multi-Broadcast With Provenance

We first present an algorithm for multi-broadcast with provenance, where all nodes must be made aware of not only the source messages, but also the IDs of the sources they originated from.

The idea of the algorithm is essentially to conduct k simultaneous binary searches to allow a leader to ascertain the IDs of all sources. The process consists of $\log L$ rounds, one for each bit of the IDs. Each node will maintain a list of known prefixes of source IDs, and we aim to preserve the invariant that, after round i, all nodes know the first i bits of every source ID. We denote the number of distinct known prefixes at the start of round i by k_i.

At the start of round i, sources know k_i distinct $i-1$-bit ID prefixes (note k_i may be less than k, since some IDs may share prefixes), and they will each construct a $2k_i$-bit string in which each bit corresponds to a particular i-bit prefix. Specifically, if we denote the known prefixes in lexicographical order by $(p_1, p_2, \dots, p_{k_i})$, then bit $2j$ in the new string will represent the prefix $p_j\mathbf{0}$, and bit $2j+1$ will represent $p_j\mathbf{1}$. Each source constructs its string by placing a **1** in the position corresponding to its own ID's i-bit prefix, and **0** in all others. We will denote the string constructed in this manner by source s in round i by $Z_{s,i}$

Performing CollectMessages with these strings ensures that the leader receives the **OR**-superimposition, which informs it of all i-bit prefixes of source IDs (since it is aware of which prefix each position corresponds to). It then broadcasts this back out to the network via the standard beep wave procedure, and thus the invariant is fulfilled round i. After $\log L$ rounds, the IDs of all sources are known in entirety by all nodes. We then perform one final CollectMessages procedure, this time to collate all of the messages the sources wish to broadcast to the network. We construct a $k \log M$-bit string in which the j^{th} block of $\log M$ bits corresponds to the message of the j^{th} source (in lexicographical order of ID). Each source individually fills in its own message in the appropriate block, leaving all other bits as **0**. We denote the string constructed in this manner by source s as $\tilde{m}_s$. Performing CollectMessages on these strings ensures that the full string of messages arrives at the leader, who then broadcasts it back out to the network.

Algorithm 10 Multi-Broadcast With Provenance$(S, m(S))$

```
$v \leftarrow$ ElectLeader
$D \leftarrow$ EstimateDiameter$(v)$
$\log M \leftarrow$ GetMessageLength$(v, S, m(S))$
for $i = 1$ to $\log L$ do
    $Z_i \leftarrow$ CollectMessages$(v, S, Z_{S,i}, 2k_i)$
    perform Beep-Wave$(v, Z_i)$
end for
$\tilde{m} \leftarrow$ CollectMessages$(v, S, \tilde{m}_S, k \cdot logM)$
perform Beep-Wave$(v, \tilde{m})$
```

▶ **Theorem 9.** Multi-Broadcast With Provenance$(S, m(S))$ *correctly performs multi-broadcast with provenance within* $O(k \log \frac{LM}{k} + D \log L)$ *time-steps.*

Proof. The three sub-procedure calls in initial "set-up" phase take a total of $O(D \log L + \log M)$ time-steps, and provide a leader node and knowledge of D and $\log M$.

Round i of the main loop of the algorithm takes $O(D + k_i)$ time, since it consists of performing CollectMessages on strings of length $O(k_i)$, and then Beep-Wave on a string of the same length. Furthermore, since the number of known prefixes at most doubles each round, $k_i \leq 2^{i-1}$. Hence, there exists some constant c such that total time for the loop is bounded by:

$$\begin{aligned}\sum_{i=1}^{\log L} c(D+k_i) &= cD\log L + c\left(\sum_{i=1}^{\log k} k_i + \sum_{i=\log k+1}^{\log L} k_i\right)\\ &\leq cD\log L + c\left(\sum_{i=1}^{\log k} 2^{i-1} + \sum_{i=\log k+1}^{\log L} k\right)\\ &\leq cD\log L + c(k + k(\log L - \log k))\\ &= O(D\log L + k\log\frac{L}{k}) \ .\end{aligned}$$

The final call to CollectMessages then takes a further $O(D + k\log M)$ time, and so total running time is $O(D\log L + k\log\frac{L}{k} + k\log M) = O(k\log\frac{LM}{k} + D\log L)$

Correctness follows since each round of the loop informs the leader of the next bit in each ID prefix, and it then broadcasts this information to the network. After $\log L$ rounds, all nodes know all source IDs and each source s can correctly construct its string $\tilde{m}_s$. The **OR**-superimposition of these strings, broadcast to all nodes, is a list of messages in source ID order, which fulfills the goal of the algorithm. ◀

We remark that this result yields an algorithm for gossiping with running time $O(n\log\frac{LM}{n} + D\log L)$, slightly improving over the $O(n\log LM)$ running time of Algorithm 4.

6.2 Multi-Broadcast Without Provenance

It may be the case that we do not need to know where messages originated from, or the number of duplicate messages; for example when using short control messages instructing all nodes to perform some action, for which provenance might be irrelevant. For this reason, we also study the variant of multi-broadcast where nodes need only know one copy of each unique source message, and no source IDs.

The main difference in concept for our multi-broadcast without provenance algorithm (Algorithm 11) is that the concurrent binary searches are performed on the bits of the source messages rather than the IDs. However, this turns out to be slower when k is smaller than D, and so we first run Algorithm 10, curtailing it when our number k_i of known ID prefixes (which is a lower bound for k) exceeds D, in order to efficiently deal with these cases.

If $k \leq D$ then the call to Algorithm 10 will complete multi-broadcast (meeting the requirements for the case without provenance, since they are strictly weaker than those with provenance). Otherwise, we move onto performing binary searches on the bits of the message. This functions in much the same way as in Algorithm 10, except that we do not need the final CollectMessages and Beep-Wave stage since the network is already aware of all source messages upon completion of the main loop. We will use $\tilde{k}_i$ to denote the number of $i-1$-bit message prefixes known to nodes at the start of round i of the for loop, and $\tilde{Z}_{s,i}$ to be the string constructed by source s in round i by placing a **1** in the position corresponding to the i-bit prefix of its message and **0** in all others.

▶ **Theorem 10.** Multi-Broadcast Without Provenance$(S, m(S))$ *correctly performs multi-broadcast without provenance within* $O(k\log\frac{M}{k} + D\log L)$ *time-steps if* $k < M$, *and* $O(M + D\log L)$ *time-steps if* $k \geq M$.

Proof. By the same argument as for Theorem 10, each round of the main loop informs all nodes of the next bit in each message prefix. Therefore, after $\log M$ rounds we have performed multi-broadcast without provenance.

Algorithm 11 MULTI-BROADCAST WITHOUT PROVENANCE($S, m(S)$)

```
perform MULTI-BROADCAST WITH PROVENANCE($S, m(S)$) until $k_i > D$
if it did not complete then
    for $i = 1$ to $\log M$ do
        $\tilde{Z}_i \leftarrow$ COLLECTMESSAGES($v, S, \tilde{Z}_{S,i}, 2\tilde{k}_i$)
        perform BEEP-WAVE($v, \tilde{Z}_i$)
    end for
end if
```

We separate the running-time proof into four cases:

1. $k \le D$ and $k < M$.
2. $k \le D$ and $k \ge M$.
3. $k > D$ and $k < M$.
4. $k > D$ and $k \ge M$.

Case 1: $k \le D$ and $k < M$. For the $k \le D$ case, the number of unique i-bit source ID prefixes k_i will never exceed D (since it is bounded above by k), and so the call to MULTI-BROADCAST WITH PROVENANCE will successfully perform multi-broadcast (with provenance, and therefore also without) in $O(k \log \frac{LM}{k} + D \log L) = O(k \log L + k \log \frac{M}{k} + D \log L) = O(k \log \frac{M}{k} + D \log L)$ time-steps.

Case 2: $k \le D$ and $k \ge M$. As above, the call to MULTI-BROADCAST WITH PROVENANCE will successfully perform multi-broadcast in $O(k \log \frac{LM}{k} + D \log L) = O(k \log L + D \log L) = O(D \log L)$ time-steps.

Case 3: $k > D$ and $k < M$. Since $k > D$, the call will not complete multi-broadcast, but its “set-up” phase will provide a leader v and knowledge of D and $\log M$, so these steps are not duplicated in our description of Algorithm 11. Each round of the main loop then informs every node of the next bit in each unique message prefix, and so after $\log M$ rounds we are done.

Let t be the round of the loop at which the call to MULTI-BROADCAST WITH PROVENANCE terminates. Running time for the call is then bounded above (for some constant c) by

$$cD \log L + \sum_{i=1}^{t} c(D+k_i) \le cD \log L + \sum_{i=1}^{t} 2cD \le cD \log L + \sum_{i=1}^{\log L} 2cD = 3cD \log L = O(D \log L) \ ,$$

where the first inequality is due to the fact that $k_i \le D$ until termination.

Running time for the main loop of Algorithm 11 is bounded above (again for some constant c) by:

$$\begin{aligned}\sum_{i=1}^{\log M} c(D+\tilde{k}_i) &= cD \log M + c\left(\sum_{i=1}^{\log k} \tilde{k}_i + \sum_{i=\log k+1}^{\log M} \tilde{k}_i\right) \\ &\le cD \log M + c\left(\sum_{i=1}^{\log k} 2^{i-1} + \sum_{i=\log k+1}^{\log M} k\right) \\ &\le cD \log M + c(k + k(\log M - \log k)) = O(D \log M + k \log \frac{M}{k}) \ .\end{aligned}$$

Total time is therefore

$$\begin{aligned} O(D\log L + D\log M + k\log\frac{M}{k}) &= O(D\log L + D\log\frac{M}{k} + D\log k + k\log\frac{M}{k}) \\ &= O(k\log\frac{M}{k} + D\log L)\ , \end{aligned}$$

where the last expression holds since $D\log k \le D\log L$ and $D\log\frac{M}{k} \le k\log\frac{M}{k}$.

Case 4: $k > D$ and $k \ge M$. The call to Multi-Broadcast With Provenance will fail and take $O(D\log L)$ time as before. Running time for the main loop of Algorithm 11 is now bounded by:

$$\begin{aligned} \sum_{i=1}^{\log M} c(D+\tilde{k}_i) &= cD\log M + c\sum_{i=1}^{\log M}\tilde{k}_i \le cD\log M + c\sum_{i=1}^{\log M} 2^{i-1} \\ &\le cD\log M + cM = O(D\log M + M)\ . \end{aligned}$$

Since $M \le k \le L$, total running time is $O(M + D\log L)$.

Combining cases

When $M > k$ total running time is $O(k\log\frac{M}{k} + D\log L)$, and when $M \le k$, total running time is $O(M + D\log L)$. ◀

It may seem unintuitive that Algorithm 11 achieves multi-broadcast in fewer then the $k\log M$ time-steps required for a single node to directly transmit or hear the messages, since this might seem to be a natural lower bound. The improvement stems from implicit compression of the messages within the algorithm's method. We next prove lower bounds that match our algorithmic results, modulo the $D\log L$ additive term.

6.3 Lower Bounds

In this section we give lower bounds for the problem of multi-broadcasting. The proofs of these bounds assume that k is greater than 1; the $k = 1$ case follows from the lower bounds given for broadcasting.

▶ **Theorem 11.** *Any algorithm achieving multi-broadcast with provenance must require $\Omega(k\log\frac{LM}{k} + D)$ time-steps.*

Proof. D is an obvious lower bound, since information can only be propagated via beeps one adjacency layer per time-step. Hence, if nodes u and v are the furthest pair in the network, D time-steps are required for v to receive any information from u. When u is a source, this means that at least D time-steps are required before v can know its message, which is necessary for multi-broadcast to be successful.

Consider any node w. By the end of the multi-broadcast algorithm, w must be aware of all (source ID, source message) pairs. It may be that w is itself a source, and already knows its own pair, but it must still learn all $k-1$ others. The number of possibilities for this $k-1$-size set of pairs is $\binom{L-1}{k-1}\cdot M^{k-1}$, since IDs must be unique but messages need not be. The number of bits required to distinguish one particular case is at least:

$$\begin{aligned} \log\left(\binom{L-1}{k-1}\cdot M^{k-1}\right) &\ge \log\left(\frac{(L-1)M}{k-1}\right)^{k-1} = (k-1)\log\left(\frac{(L-1)M}{k-1}\right) \\ &\ge \frac{k}{2}\log\left(M\sqrt{\frac{L}{k}}\right) \ge \frac{1}{4}k\log\frac{LM}{k}\ . \end{aligned}$$

Here we used the inequality $\frac{L-1}{k-1} \geq \sqrt{\frac{L}{k}}$, which is true whenever $L \geq k > 1$.

Since a node can only receive at most one bit of information per time-step, $\frac{1}{4}k \log \frac{LM}{k}$ time-steps are required for it to receive all of the information necessary for multi-broadcast with provenance. Therefore, total time required is at least $\max(D, \frac{1}{4}k \log \frac{LM}{k}) \geq \frac{1}{8}(D + k \log \frac{LM}{k}) = \Omega(k \log \frac{LM}{k} + D)$. ◀

▶ **Theorem 12.** *Any algorithm achieving multi-broadcast without provenance must require $\Omega(k \log \frac{M}{k} + D)$ time-steps if $M > k$, or $\Omega(M + D)$ if $M \leq k$. We assume that $M > 1$.*

Proof. As before, D is an obvious lower bound.

We will again consider the amount of information a node w must receive for multi-broadcast to be successful. Node w must become aware of the set of all source messages, and it starts with knowledge of at most 1 (its own, if it is a source). The number of possibilities for the remaining messages is at least $\sum_{i=0}^{k-1} \binom{M-1}{i}$, since any subset of size at most $k-1$ of the remaining message space is possible (messages need not be unique, so there need not be exactly $k-1$ other messages). To distinguish a particular case, the number of bits of information w must receive is at least $\log \left(\sum_{i=0}^{k-1} \binom{M-1}{i} \right)$.

If $M > k$, then we have the following:

$$\log \left(\sum_{i=0}^{k-1} \binom{M-1}{i} \right) \geq \log \binom{M-1}{k-1} \geq \log \left(\frac{M-1}{k-1} \right)^{k-1} \geq (k-1) \log \left(\sqrt{\frac{M}{k}} \right)$$
$$\geq \frac{1}{4} k \log \left(\frac{M}{k} \right) \quad .$$

Similarly to the proof of Theorem 11, we used that $\frac{M-1}{k-1} \geq \sqrt{\frac{M}{k}}$ for $M \geq k > 1$.

If $M \leq k$, then we have the following:

$$\log \left(\sum_{i=0}^{k-1} \binom{M-1}{i} \right) = \log(2^{M-1}) = M - 1 \geq \frac{M}{2} \quad .$$

So, if $M > k$ then w must receive at least $\frac{1}{4} k \log \frac{M}{k}$ bits, and therefore $\max(D, \frac{1}{4}k \log \frac{M}{k}) \geq \frac{1}{8}(D + k \log \frac{M}{k}) = \Omega(k \log \frac{M}{k} + D)$ time-steps are required.

If $M \leq k$, then w must receive at least $\frac{M}{2}$ bits, and therefore $\max(D, \frac{M}{2}) \geq \frac{1}{4}(D + M) = \Omega(M + D)$ time-steps are required. ◀

The assumption that $M > 1$ is essential; in the case that $M = 1$ and $k = n$, all nodes know the only possible message already, and a multi-broadcast without provenance algorithm can terminate immediately rather than requiring the $\Omega(D)$ time-steps the lower bound would imply.

7 Conclusion

The beep model is interesting as a widely applicable model that requires very little of communications devices, and can be applied even where restrictive circumstances frustrate communication under more complex models. Furthermore, it is an interesting technical challenge to design efficient algorithms while making the minimum possible assumptions about the network. In this paper we have given deterministic algorithms for several fundamental communications tasks in the beep model. The model is still young, however, and there are many remaining avenues for fruitful research.

The most pressing question our work here raises is whether the $D \log L$ additive term in the running time of our multi-broadcasting algorithms can be reduced to D, or whether the algorithms are in fact optimal in all cases. Since $D \log L$ is the cost of leader election, a fundamental starting point for our algorithms, any improvement would have to begin here. However, a faster leader election algorithm alone would not be sufficient to improve multi-broadcast time, since it is not a bottleneck in our algorithms; they also require $D \log L$ time elsewhere. This may suggest that $D \log L$ is indeed a lower bound, and again, it may be wisest to focus on leader election to prove this. Multi-broadcast with provenance and constant M is at least as hard as leader election from a set of k candidates, since after performing the multi-broadcast with the candidates as sources, their IDs (and in particular the highest ID) are known to all nodes.

A different possible focus for further research is to determine to what extent randomization can help. The leader election algorithm of [6], taking $O((D + \log n \log \log n) \cdot \min(\log \log n, \log \frac{n}{D}))$ time and succeeding with high probability, demonstrates that improvements over deterministic algorithms can be made. It seems likely that randomization could also be of use in algorithms for multi-broadcast, though as mentioned, simply employing this randomized leader election algorithm rather than the deterministic one does not reduce the asymptotic running time of our multi-broadcast algorithms.

References

1 Y. Afek, N. Alon, Z. Bar-Joseph, A. Cornejo, B. Haeupler, and F. Kuhn. Beeping a maximal independent set. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, pages 32–50, 2011.

2 M. Chrobak, L. Gąsieniec, and W. Rytter. Fast broadcasting and gossiping in radio networks. *Journal of Algorithms*, 43(2):177–189, 2002.

3 A. Cornejo and F. Kuhn. Deploying wireless networks with beeps. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, pages 148–262, 2010.

4 K.-T. Förster, J. Seidel, and R. Wattenhofer. Deterministic leader election in multi-hop beeping networks. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*, pages 212–226, 2014.

5 M. Ghaffari, B. Haeupler, , and M. Khabbazian. Randomized broadcast in radio networks with collision detection. In *Proceedings of the 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–334, 2013.

6 M. Ghaffari and B. Haeupler. Near optimal leader election in multi-hop radio networks. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 748–766, 2013.

7 S. Gilbert and C. Newport. The computational power of beeps. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC)*, pages 31–46, 2015.

8 K. Hounkanli and A. Pelc. Deterministic broadcasting and gossiping with beeps. In arxiv 1508.06460, 2015.

9 D. Peleg. Time-efficient broadcasting in radio networks: A review. In *Proceedings of the 4th International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 1–18, 2007.

Nontrivial and Universal Helping for Wait-Free Queues and Stacks

Hagit Attiya[*1], **Armando Castañeda**[†2], **and Danny Hendler**[3]

1 **Technion – Israel Institute of Technology, Haifa, Israel**
2 **Universidad Nacional Autónoma de México (UNAM), Mexico City, Mexico**
3 **Ben-Gurion University, Beer-Sheva, Israel**

Abstract

This paper studies two approaches to formalize helping in wait-free implementations of shared objects. The first approach is based on *operation valency*, and it allows us to make the important distinction between *trivial* and *nontrivial* helping. We show that a wait-free implementation of a queue from common2 objects (e.g., Test&Set) requires nontrivial helping. In contrast, there is a wait-free implementation of a stack from Common2 objects with only trivial helping. This separation might shed light on the difficulty of implementing a queue from Common2 objects.

The other approach formalizes the helping mechanism employed by Herlihy's universal wait-free construction and is based on having an operation by one process restrict the possible linearizations of operations by other processes. We show that objects possessing such *universal helping* can be used to solve consensus.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases helping, wait-free, nonblocking, queues, stacks, common2

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.31

1 Introduction

A key component in the design of concurrent applications are *shared objects* providing higher-level semantics for communication among processes. For example, a *shared queue* to which processes can concurrently enqueue and dequeue, allows them to share tasks, and similarly a *shared stack*. Shared objects are implemented from more basic *primitives* supported by the multiprocessing architecture, e.g., reads, writes, Test&Set, or Compare&Swap. An implementation is *wait-free* if an operation on the shared object is guaranteed to terminate after a finite number of steps; the implementation is *nonblocking* if it only ensures that *some* operation (perhaps by another process) completes in this situation. Clearly, a wait-free implementation is nonblocking but not necessarily vice versa.

Many implementations of shared objects, especially the wait-free ones, include one process *helping* another process to make progress. The helping mechanism is often some code that is added to a nonblocking implementation. Typically, the code uses only reads and writes, in addition to the primitives used in the nonblocking implementation (e.g., Test&Set). The aim of this extra code is that processes that complete an operation "help" the blocked processes to terminate, so that the resulting implementation is wait-free. An interesting

[*] Supported by the Israel Science Foundation (grant 1749/14).
[†] Supported partially by PAPIIT-UNAM IA101015. This research was partially done while the second author was at the Department of Computer Science of the Technion, supported by an Aly Kaufman post-doctoral fellowship.

example is a shared queue, for which there is a simple nonblocking implementation using only reads, writes and Test&Set [14]. Such a helping mechanism would provide a wait-free queue implementation using those primitives, showing that the queue belongs to Common2, the family of shared objects that are wait-free implementable from primitives with consensus number 2 for any number of processes [1, 2]. The Common2 family contains Test&Set, Swap, stacks and other objects.

The question whether queues belong to Common2 has been open for many years and has received a considerable amount of attention [2, 4, 5, 6, 8, 14]. It essentially asks if there is an n-process linearizable wait-free implementation of a queue from Test&Set, for every $n \geq 3$.

This paper investigates ways to formalize helping, with the purpose of being able to separate objects for which there are wait-free implementations from those with only nonblocking implementations. We are especially interested in implementations using primitives with finite *consensus number* [12], like Test&Set, which allows us to solve consensus exactly for two processes. Primitives with an infinite consensus number, like Compare&Swap, are universal and provide generic wait-free implementation for any shared object [12]; clearly, with such primitives nonblocking and wait-freedom cannot be separated.

We first introduce a notion of helping that is based on one process determining the return value of an operation by another process; it relies on the notion of *operation valency* [11], i.e., the possible values an operation might return. Roughly speaking, an implementation has helping if in some situation, a process makes an undecided operation of another process become decided on some value. In the context of specific objects, like queues and stacks, which have a distinguished "empty" value (denoted $\perp$), we say that helping is *nontrivial* if one process makes another become decided on a non-$\perp$ value. Helping is nontrivial since the helping process needs to "grab" the value it gives to the helped process. Therefore, the helping process has to communicate with the other processes to ensure that this value is not taken by someone else.

Our first main result is a separation between stacks and queues implemented from Test&Set. It shows that any wait-free queue must have nontrivial helping while this is not true for stacks, as we show that the wait-free stack of Afek et al. [1] (which established that stacks belong to Common2) does not have nontrivial helping.

The paper also studies an alternative way to formalize helping, which is based on restricting the possible linearizations of an operation by the progress of another process. This kind of helping, which we call *universal*, formalizes the helping mechanism employed in Herlihy's universal construction. Intuitively, an implementation has universal helping if for every execution α, for every long enough extension of it, all pending operations in α (which might be still pending in the extension) are linearized. We show that universal helping for queues and stacks is strong enough to solve consensus, namely, any wait-free n-process implementation of a queue or stack with universal helping can solve n-process consensus.

These results provide insights on why finding a wait-free implementation for queues from Test&Set has been a longstanding open question: any such implementation must have some helping mechanism; however, this mechanism cannot be too strong, otherwise the resulting implementation would be able to solve consensus for n processes, $n \geq 3$, which is impossible to do with Test&Set since it has consensus number 2.

Finally, the paper compares the two formalizations proposed here to the formalization of helping recently introduced in [3]. It also shows that universal helping has implications on *strong linearizability* [9] for queues and stacks: there is no n-process wait-free strong linearizable implementation of queues or stacks from primitives with consensus number smaller than n.

2 Model of Computation

We consider a system with n *asynchronous* processes, $p_1, \ldots, p_n$. Processes communicate with each other by applying *primitives* to shared *base objects*; the primitives can be read and write, or more powerful primitives like Test&Set or Compare&Swap. Any process may crash at any time in an execution, namely, it stops taking steps from that point on. A process that does not crash is *correct*.

A *(high-level) concurrent object*, or *data type*, is defined by a state machine consisting of a set of states, a set of operations, and a set of transitions between states. Such a specification is known as *sequential*. In the rest of the paper we will concentrate on stacks and queues. A *shared stack* provides two operations push($\cdot$) and pop(). A push(x) operation puts x at the top of the stack, and a pop() removes and returns the value at the top, if there is one, otherwise it returns $\perp$. A *shared queue* provides operations enq($\cdot$) and deq(). An enq(x) operation puts x at the tail of the queue, and a deq() removes and returns the value at the head of the queue, if there is one, otherwise it returns $\perp$.

An *implementation* of an object O is a distributed algorithm $\mathcal{A}$ consisting of local state machines $A_1, \ldots, A_n$. Local machine A_i specifies which primitives p_i executes in order to return a response when it invokes an operation of O. An implementation is *wait-free* if every process completes each of its invocations in a finite number of its steps. Formally, if a process executes infinitely many steps in an execution, it completes all its invocations. An implementation is *nonblocking* if whenever processes take steps, at least one of the operations terminates. Namely, in every infinite execution, infinitely many invocations are completed. Thus, a wait-free implementation is nonblocking but not necessarily vice versa.

A *configuration* C of the system is a collection containing the states of all base objects and processes. A configuration is *initial* if base objects and processes are in initial states. Given a configuration C, for any process p, $p(C)$ denotes the configuration after p takes its next step. A process p is *idle* in a configuration C if p is in a state in which all its operations are completed.

An *execution* of the system is modelled by a *history*, which is a possibly infinite sequence of invocations and responses of high-level operations and primitives. For a set of processes S, an S-*execution* is an execution in which only processes in S take steps. If $S = \{p\}$, we say that the execution is p-*solo*. An operation op in a history is *complete* if both its invocation inv(op) and response res(op) appear in the history. An operation is *pending* if only its invocation appears in the history.

A history H induces a natural partial order $<_H$ on the operations of H: op $<_H$ op$'$ if and only if res(op) precedes inv(op$'$). Two operations are *concurrent* if they are incomparable. A *sequential* history alternates matching invocations and responses and starts with an invocation event. Hence, if H is sequential, $<_H$ induces a total order.

Linearizability [13] is the standard notion used to identify a correct implementation. Roughly speaking, an implementation is linearizable if each operation appears to take effect atomically at some time between the invocation and response of an operation.

Let $\mathcal{A}$ be an implementation of an object O. A history H of $\mathcal{A}$ is *linearizable* if H can be extended by adding response events for some pending invocations such that the sequence H' containing only the invocation and responses of O agrees with the specification of O, namely, there is an initial state of O and a sequence of invocations and responses that produces H'. We say that $\mathcal{A}$ is *linearizable* if each of its histories is linearizable.

In the *consensus* problem, each process proposes a value and is required to decide on a value such that the following properties are satisfied in every execution:

Termination. Every correct process decides.
Agreement. Processes decide on the same value.
Validity. Processes decide proposed values.

Consensus is *universal* [12] in the sense that from reads and writes and objects solving consensus among n processes, it is possible to obtain a wait-free implementation for n processes of any concurrent object with a sequential specification. The *consensus number* of a primitive [12] is the maximum number n such that it is possible to solve consensus on n processes from reads, writes and the primitive. For example, the consensus number of Test&Set is 2. Hence, Test&Set allows us to implement any concurrent object in a system with 2 processes.

3 Separating Stacks and Queues with Nontrivial (Valency-Based) Helping

In this section, we present a notion of helping that differentiates between queues and stacks: any queue implementation must exhibit this kind of helping, but there is a stack implementation that does not (essentially, that of [1]). This sheds some light on the difficulty of finding a wait-free implementation of a queue from Common2.

Let $\mathcal{A}$ be a wait-free linearizable implementation of a data type T, such as a stack or queue. The input for an invocation of an operation of T is from some domain $\mathcal{V}$ and the output of a response is from the domain $\mathcal{V} \cup \{\perp\}$, where $\perp \notin \mathcal{V}$ denotes the *empty* or *initial* state of T.

Let C be a reachable configuration of $\mathcal{A}$ and let $\mathsf{opType}(\cdot)$ be an operation by a process p. We say that $\mathsf{opType}(\cdot)$ is *v-univalent in C* (or just *univalent* when v is irrelevant) if in every configuration C' that is reachable from C in which $\mathsf{opType}(\cdot)$ is complete, its output value is v; otherwise, $\mathsf{opType}(\cdot)$ is *multivalent in C*. We say that $\mathsf{opType}(\cdot)$ is *critical on v in C* (or just *critical in C*) if it is multivalent in C but v-univalent in $p(C)$.

▶ **Definition 1** (Nontrivial and trivial (valency-based) helping)**.** Process q *helps* process $p \neq q$ in configuration C if there is a multivalent $\mathsf{opType}(\cdot) \in C$ by p that is v-univalent in $q(C)$. We say that q *nontrivially helps* p if $v \neq \perp$; otherwise, it *trivially* helps p.
An implementation of a type T has *nontrivial (trivial) helping* if it has a reachable configuration C such that some process q nontrivially (trivially) helps process p in C.

Directly from the previous definition we get the following claim.

▶ **Claim 2.** *If C is a reachable configuration of an algorithm without nontrivial helping, and an operation* op *by p is multivalent in C, then* op *is not v-valent in $q(C)$, for any value $v \neq \perp$ and process $q \neq p$.*

The proof of the next theorem captures the challenging "tail chasing" phenomenon one faces when trying to implement a queue from objects in Common2. Observe that in the case of a queue implementation, only dequeues can be nontrivially helped since enqueues always return true, and are therefore trivially univalent.

▶ **Theorem 3.** *Any two-process wait-free linearizable queue implementation from* read/write *and* Test&Set *operations has nontrivial helping.*

Proof. Assume, by way of contradiction, there is such an implementation $\mathcal{A}$ without nontrivial helping. Let p and q be two distinct processes, and let C_{init} be the initial configuration of $\mathcal{A}$.

For any $k \geq 1$, we construct an execution α_k of p and q, starting with C_{init} and ending in configuration C_k. In α_k, p executes a single $\mathsf{deq_p}()$ operation, and the following properties hold:

1. q is idle in C_k,
2. p has at least k steps in α_k,
3. in every linearization of α_k, all enqueues appear in the same order and enqueue distinct values,
4. there is no linearization of α_k in which $\mathsf{deq_p}()$ outputs $\bot$, and
5. $\mathsf{deq_p}()$ is multivalent in C_k (in particular, it is pending).

We proceed by induction. For the base case, $k = 1$, let α_1 be the execution that starts at C_{init} and in which p completes alone $\mathsf{enq}(1)$ and then starts $\mathsf{deq_p}()$ until it is critical on 1. This execution exists because $\mathcal{A}$ is wait-free. Clearly, there is no linearization of α_1 in which $\mathsf{deq_p}()$ outputs $\bot$. The other properties also hold.

Suppose that we have constructed α_k, $k \geq 1$; we show how to obtain α_{k+1}. Let β^1 be the q-solo extension of α_k in which q completes $\mathsf{enq}(z)$, where z is a value that is not enqueued in α_k, and then starts a $\mathsf{deq}_q()$ operation. Let β^2 be an extension of $\alpha_k\,\beta^1$ in which p and q take steps until both their dequeue operations are critical. The extension β^2 exists because, first, $\mathcal{A}$ is wait-free and, second, by Claim 2, a step of p does not make $\mathsf{deq_q}()$ univalent, and a step of q does not make $\mathsf{deq_p}()$ univalent.

Let C be the configuration at the end of $\alpha_k\,\beta^1\,\beta^2$; note that $\mathsf{deq_p}()$ is critical on some value y_p in C and that $\mathsf{deq_q}()$ is critical on some value y_q in C.

Note that neither y_p nor y_q is $\bot$ since the queue has at least two values in C. This holds since the induction hypothesis is that there is no linearization of α_k in which $\mathsf{deq_p}()$ outputs a non-$\bot$ value, and in the extension $\beta^1\,\beta^2$, q first completes an enqueue and then starts a dequeue.

By a similar argument, there is no linearization of $\alpha_k\,\beta^1\,\beta^2$ in which either $\mathsf{deq_p}()$ or $\mathsf{deq_q}()$ outputs $\bot$.

The following claim is where the specification of a queue comes into play. (This claim does not hold for a stack, for example.)

▶ **Claim 4.** $y_p = y_q$.

Proof. Suppose, by way of contradiction, that $y_p \neq y_q$. By the induction hypothesis, in every linearization of α_k, the order of enqueues is the same. The same holds for $\alpha_k\,\beta^1\,\beta^2$ because q is idle in α_k, by induction hypothesis, and then its enqueue in β^1 happens after all enqueues in α_k. Suppose, without loss of generality, that $\mathsf{enq}(y_q)$ precedes $\mathsf{enq}(y_p)$ in every linearization of $\alpha_k\,\beta^1\,\beta^2$. Consider the p-solo extension in which p completes $\mathsf{deq_p}()$, ending with configuration D.

Since $\mathsf{deq_p}()$ is critical on y_p in C, it outputs y_p in D. We claim that $\mathsf{deq_q}()$ outputs y_q in every extension from D. Otherwise, another dequeue outputs y_p in some extension from D. Since this dequeue starts after $\mathsf{deq_p}()$ completes, it must be linearized after $\mathsf{deq_p}()$. This contradicts the linearizability of $\mathcal{A}$, since in every linearization of $\alpha_k\,\beta^1\,\beta^2$, $\mathsf{enq}(y_q)$ precedes $\mathsf{enq}(y_p)$. Therefore, $\mathsf{deq_q}()$ is y_q-univalent in D. This contradicting Claim 2, since a step of p makes $\mathsf{deq_q}()$ univalent on a non-$\bot$ value. ◀

Note that there is no extension of C in which $\mathsf{deq_p}()$ and $\mathsf{deq_q}()$ output the same value because the enqueues in α_k have distinct values and in β^1, q enqueues z, a value that is not enqueued in α_k.

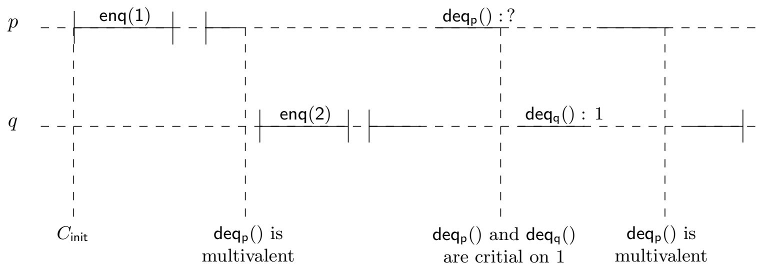


Figure 1 Getting α_2 from α_1.

Assume that p is poised to access R_p in C (i.e. the next step of p is on R_p) and that q is poised to access R_q in C. If $R_p \neq R_q$, then in the p-solo extension of $q(p(C))$ in which p completes $\mathsf{deq_p}()$, its output is $y = y_p = y_q$. But in $p(q(C))$, the local state of p and the state of the shared memory is the same as in $q(p(C))$. Hence, in the p-solo extension of $p(q(C))$, p completes $\mathsf{deq_p}()$ with output y, as well. This contradicts the fact that $\mathsf{deq_q}()$ is critical on y in C. Thus, $R_p = R_q = R$.

A similar argument, by case analysis, shows that p and q must apply Test&Set primitives to R in C, and that the value of R is 0 in C. These facts are used in the proof of the next claim.

▶ **Claim 5.** $\mathsf{deq_p}()$ *is not critical in* $q(C)$.

Proof. Let $y = y_p = y_q$ be the value that $\mathsf{deq_p}()$ and $\mathsf{deq_q}()$ are critical on in C. Suppose, by way of contradiction, that $\mathsf{deq_p}()$ is critical on y' in $q(C)$. We have that $y' \neq y$.

Let γ be an extension of $\alpha_k\, \beta^1\, \beta^2\, q$ in which $\mathsf{deq_p}()$ outputs. Write $\gamma = \lambda^1\, p\, \lambda^2$, where λ^1 is p-free (λ^1 might be empty). Since p and q are about to perform Test&Set primitives on R in C, the state of the shared memory and the local state of p are the same at the end of $\alpha_k\, \beta^1\, \beta^2\, q\, \lambda^1\, p$ and $\alpha_k\, \beta^1\, \beta^2\, q\, p\, \lambda^1$, because in both executions q is the first process accessing R (using Test&Set) and then when p accesses R (using Test&Set also), it gets false, no matter when it accesses R. Then, p is in the same local state in $\alpha_k\, \beta^1\, \beta^2\, q\, \lambda^1\, p\, \lambda^2$ and in $\alpha_k\, \beta^1\, \beta^2\, q\, p\, \lambda^1\, \lambda^2$. We have that $\mathsf{deq_p}()$ is critical in $q(C)$, which implies that the output of it in $\alpha_k\, \beta^1\, \beta^2\, q\, p\, \lambda^1\, \lambda^2$ is y', and thus the output of $\mathsf{deq_p}()$ in $\alpha_k\, \beta^1\, \beta^2\, q\, \lambda^1\, p\, \lambda^2$ is y' too, since, as already said, the local state of p is the same in both executions. This implies that $\mathsf{deq_p}()$ is univalent in $q(C)$, contrary to our assumption that it is critical in $q(C)$. ◀

Let $\alpha_{k+1} = \alpha_k\, \beta^1\, \beta^2\, q\, p\, \beta^3$, where β^3 is the q-solo extension in which q completes its $\mathsf{deq}_q()$ (β^3 exists because $\mathcal{A}$ is wait-free). See Figure 1. We argue that α_{k+1} has the desired properties.

1. q is idle in α_{k+1} because in β^3 it completes $\mathsf{deq_q}()$ and does not start a new operation.
2. p has at least $k + 1$ steps of $\mathsf{deq_p}()$ in α_{k+1}, since p has at least k steps of $\mathsf{deq_p}()$ in α_k, by the induction hypothesis, and at least one step in $\beta^1\, \beta^2\, q\, p\, \beta^3$.
3. By the induction hypothesis, in every linearization of α_k, the enqueue operations follow the same order. The enqueue of q in β^1 happens after all enqueues in α_k. Then, in every linearization of α_{k+1}, the enqueues follow the same order. The enqueues in α_{k+1} enqueue distinct values because that is true for α_k, by the induction hypothesis, and the enqueue of q in β^1 enqueues a value that is not in α_k.
4. As argued above, there is no linearization of $\alpha_k\, \beta^1\, \beta^2$ in which either $\mathsf{deq_p}()$ or $\mathsf{deq_q}()$ output $\perp$. In β^3, q just completes $\mathsf{deq_q}()$. Then, there is no linearization of α_{k+1} in which $\mathsf{deq_p}()$ outputs $\perp$.

```
Shared Variables:
        $range$ : Fetch&Add register initialized to 1
        $items$ : array $[1, \ldots, ]$ of read/write registers
        $T$ : array $[1, \ldots, ]$ of Test&Set registers

Operation Push($x$):
(01)    $i$ = Fetch&Add($range$, 1)
(02)    $items[i] \leftarrow x$
        return  true
end Push

Operation Pop():
(03)    $t$ = Fetch&Add($range$, 0)
        for $i \leftarrow t$ downto 1 do
(04)        $x \leftarrow items[i]$
(05)        if $x \neq \perp$ then
(06)            if Test&Set($T[i]$) then return $x$
            end if
        end for
(07)    return $\perp$
end Pop
```

Figure 2 The stack implementation of Afek et al. [1].

5. Since $\mathsf{deq_p}()$ is not critical in $q(C)$, it is multivalent at the end of $\alpha_k\, \beta^1\, \beta^2\, q\, p$. Since β^3 is q-solo, Claim 2 implies that $\mathsf{deq_p}()$ is multivalent at the end of α_{k+1}.

This yields an execution of $\mathcal{A}$ in which p executes an infinite number of steps but its $\mathsf{deq_p}()$ operation does not complete, contradicting the wait-freedom of $\mathcal{A}$. ◀

As we just saw, any wait-free implementation of a queue from Test&Set must have nontrivial helping. This is not the case for stack implementations, as we show next.

▶ **Theorem 6.** *There is an n-process wait-free linearizable stack implementation from read/write and m-process Test&Set primitives, $2 \leq m \leq n$, without nontrivial helping.*

Proof. First, we show that an n-process wait-free linearizable Test&Set operation can be implemented from 2-process Test&Set without nontrivial helping. [2] present an n-process wait-free linearizable implementation of a Test&Set operation from 2-process one-shot swap primitives. Let us call this algorithm $\mathcal{A}$. It is easy to check that $\mathcal{A}$ does not have nontrivial helping. It is also easy to implement one-shot 2-process swap from 2-process Test&Set without helping (the processes just use Test&Set to decide who swaps first), and hence, from $\mathcal{A}$ we can get an n-process wait-free linearizable implementation of a Test&Set operation from 2-process Test&Set without nontrivial helping. Let us call the resulting algorithm $\mathcal{B}$.

Now, consider Afek et al.'s stack implementation [1] (Figure 2). We argue that the implementation does not have nontrivial helping: just note that there is no configuration C in which process q makes another process p v-univalent, $v \neq \perp$, because the only way a pop operation becomes univalent on a non-$\perp$ value is by winning the Test&Set in line 6; thus, it is impossible that a multivalent pop operation by p in C becomes univalent on a non-$\perp$ value in $q(C)$, with $q \neq p$.

Afek et al. proved that one can get an n-process wait-free linearizable Fetch&Add from 2-process wait-free linearizable Test&Set primitives [2]. Let $\mathcal{C}$ be this implementation.

Now, we replace each Test&Set primitive in Afek et al.'s implementation in Figure 2 with an instance of $\mathcal{B}$, and each Fetch&Add with an instance of $\mathcal{C}$. Let $\mathcal{A}$ be the resulting implementation. Clearly, $\mathcal{A}$ is an n-process wait-free linearizable implementation of a stack.

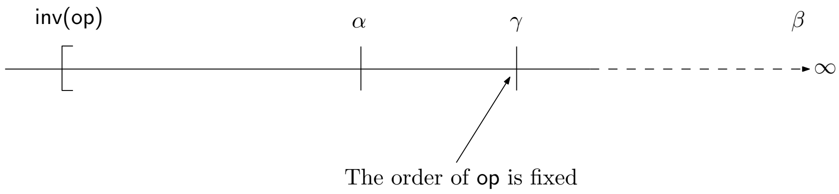


Figure 3 Universal helping: every pending operation is eventually linearized.

Moreover, it has no helping because, as mentioned already, $\mathcal{B}$ has no helping (in the sense described above) and Afek et al.'s stack implementation has no helping as well. Note that it does not matter if $\mathcal{C}$ has helping or not (trivial or nontrivial) because, as already pointed out, the only way a pop operation can become univalent is by winning the Test&Set in line 6, hence the $\mathcal{C}$ cannot change this. ◀

From Theorems 3 and 6, we get that nontrivial helping is a distinguishing factor between stacks and queues: while a stack can be implemented without nontrivial helping from read/write and Test&Set, any implementation of a queue from the same primitives necessitates nontrivial helping. Although the stack implementation of Theorem 6 is without *nontrivial* helping, it *does* have trivial helping. An example is when a process p reads the counter *range* in line 3 when there is only a single non-$\perp$ value in $T[1,\dots,t]$ (where t is the value that p reads), and then a process $q \neq p$ reads *range* after p and takes the only non-$\perp$ value in $T[1,\dots,t]$ (namely, q overtakes p). When q wins in line 6, it makes p's pop operation $\perp$-univalent because p will scan the range $T[1,\dots,t]$ without seeing any non-$\perp$ value, and will therefore return $\perp$ in line 7.

4 Universal (Linearization-Based) Helping

In this section we propose another formalization of helping, in which a process ensures that operations by other processes are eventually linearized. This definition captures helping mechanisms such as the one used in Herlihy's universal wait-free construction [12]. We evaluate the power of this helping mechanism via consensus and compare it with the valency-based helping notion studied in Section 3.

Throughout this section, we assume, without loss of generality, that the first step of every operation is to publish its *signature* (i.e., the operation type and its parameters) to the shared-memory so that it may be helped by other processes. An operation is *pending* if it has published its signature but did not yet terminate.

▶ **Definition 7** (Universal (linearization-based) Helping)**.** Consider an n-process wait-free linearizable implementation of a data type T. The implementation has *universal helping* if every infinite extension $\alpha\beta$ of a finite history α has a finite prefix γ with a linearization $\mathsf{lin}(\gamma)$, which satisfies the following conditions:

- $\mathsf{lin}(\gamma)$ contains every pending high-level operation of α (see Figure 3), in addition to all high-level operations that complete in γ.
- Every extension $\gamma\lambda$ of γ has a linearization $\mathsf{lin}'(\gamma\lambda)$ such that $\mathsf{lin}'(\gamma)$ is a prefix of it.

If the above conditions are satisfied for every γ such that some process completes $f(n)$ or more high-level operations in the extension $\gamma - \alpha$, for some function $f : \mathbb{N} \to \mathbb{N}$, then we say the implementation has *f-universal helping*.

```
Operation propose($v_i$):
(01)  $Vals[i] \leftarrow v_i$
(02)  Simulate to completion $f(n)+1$ enq($i$) operations
(03)  $S \leftarrow$ Snapshot of the shared-memory variables used by $\mathcal{B}$
(04)  $d \leftarrow$ Locally simulate a deq() operation on $\mathcal{B}$, starting from $S$
(05)  decide $Vals[d]$
endoperation
```

Figure 4 Solving consensus using $\mathcal{B}$.

Universal helping requires that the progress of some processes eventually ensures that all pending invocations are linearized and all processes make progress. f-universal helping bounds from above the number of high-level operations a process needs to perform in order to ensure the progress of other processes.

▶ **Theorem 8.** *Let $\mathcal{B}$ be an n-process nonblocking linearizable implementation of a queue or stack. If $\mathcal{B}$ has f-universal helping, then n-process consensus can be solved using $\mathcal{B}$ and read/write registers.*

Proof. First assume that $\mathcal{B}$ implements a queue. Figure 4 shows the pseudocode of an algorithm that solves consensus using $\mathcal{B}$ and read/write registers. Each process p_i first writes its proposal to $Vals[i]$ (initialized to $\perp$) in Line 01 and then performs $f(n)+1$ enq(i) operations in Line 02.

To solve consensus, p_i computes a snapshot that reads the state of the queue from the shared memory to a local variable S (Line 03) and then invokes a single deq() operation using state S in Line 04 (we say that p_i *locally simulates* the deq() operation). Finally, p_i decides (in Line 05) on the value proposed by the process whose identifier it dequeues in Line 04. The snapshot in Line 03 is taken as follows. In all executions of $\mathcal{B}$ in which each process executes at most $f(n)+1$ enqueue operations, processes access a finite set of base objects in the shared memory. Let R be the set with all base objects in all those executions. Then, processes use any read/write wait-free snapshot algorithm to take a snapshot of R. We now prove that the algorithm is correct.

Termination. Every correct process decides as each process invokes a finite number of operations of $\mathcal{B}$, which is nonblocking.

Validity. The view stored to S in Line 03 represents a state of $\mathcal{B}$ in which the queue is non-empty, since at least a single enq() operation completed and no deq() operations were invoked. Moreover, if p_i gets d from its local simulation, p_d participated in the execution. It follows that every correct process p_i decides on a proposed value.

Agreement. We prove that all correct processes dequeue the same value in Line 04, from which agreement follows easily. Let E be an execution of the algorithm of Figure 3. Let p_i, p_j be two distinct correct processes. Let α_i (resp. α_j) be the shortest prefix of E in which the first enqueue operation performed by p_i (resp. p_j) in Line 02 completes. Let γ_i (resp. γ_j) denote the shortest extension of α_i in which p_i (resp. p_j) completes the last enqueue operation in Line 02.

WLOG, assume that γ_i is a prefix of γ_j, that is, p_i is the first to complete Line 02. In γ_i, p_i completed $f(n)$ enq() operations after completing its first enqueue operation on $\mathcal{B}$. Consequently, Definition 7 guarantees that its first enq(i) operation, as well as any operations that preceded it and pending operations that were concurrent with it, are linearized in $lin(\gamma_i)$ and their order is fixed. Since no dequeue operations are applied to (the shared copy of) B, $lin(\gamma_i)$ consists of enqueue operations only. Let $\mathsf{enq}(k)$ be the first operation in $lin(\gamma_i)$.

Let β_i (resp. β_j) denote the shortest prefix of E in which p_i (resp. p_j) completes Line 04. Since γ_i is a prefix of both β_i and β_j, it follows from Definition 7 that there are linearizations $lin'(\beta_i)$ and $lin'(\beta_j)$ in which $\mathsf{enq}(k)$ is the first operation. It follows in turn that the dequeue operations of both p_i and p_j in Line 04 return k, hence they both decide on $vals[k]$ in Line 05.

A similar argument gives the same result for stacks. The difference is that in the local simulation, a process simulates pop operations until it gets an *empty* response, and then decides on the proposed value of the process whose identifier was popped last (hence, pushed first). ◀

Herlihy's universal construction [12] has f-universal helping. Thus, Theorem 8 implies that, for stacks and queues, Herlihy's construction uses the full power of n-consensus in the sense that the resulting implementations can actually be used to solve n-consensus.

The next lemma will be used to show that for queues and stacks, universal helping implies nontrivial helping.

▶ **Lemma 9.** *Let T be a data type with two operations $\mathsf{put}(x)$ and $\mathsf{get}()$ such that, for distinct processes p and q, there is an infinite sequential execution S of T containing only put operations by q with a prefix S' such that:*

P1: *For every prefix $S' \cdot S''$ of S, in every sequential extension $S' \cdot S'' \cdot \langle p.\mathsf{get}() : \mathsf{return}\, y\rangle$, $y \neq \bot$ holds, where $\bot$ is the initial state of T.*

P2: *In every pair of sequential extensions $S' \cdot \langle p.\mathsf{get}() : \mathsf{return}\, y_1\rangle \cdot \langle q.\mathsf{get}() : \mathsf{return}\, z_1\rangle$ and $S' \cdot \langle q.\mathsf{get}() : \mathsf{return}\, z_2\rangle \cdot \langle p.\mathsf{get}() : \mathsf{return}\, y_2\rangle$, $y_1 \neq y_2$ holds.*

Then, any wait-free linearizable implementation of T with universal helping also has nontrivial helping.

Proof. Consider sequential executions S and S' of T as the lemma assumes. Let $\mathcal{A}$ be a wait-free linearizable implementation of T with universal helping. Let α be an execution of $\mathcal{A}$ in which q completes alone all operations in S', in that order. We claim that a $\mathsf{get}_p()$ by p is multivalent in the configuration at the end of α (note that at the end of α, $\mathsf{get}_p()$ has not even started): by property P2, the output of $\mathsf{get}_p()$ in the extension of α in which $\mathsf{get}_p()$ is completed alone and then a $\mathsf{get}_q()$ by q is completed alone, is different from the output in which the operations are completed in the opposite order.

Now, let α' be an extension of α in which p executes alone a $\mathsf{get}_p()$ until the operation is critical. Let β be an infinite extension of α' in which q runs alone and executes the operations in $S - S'$, in that order. Since $\mathcal{A}$ has universal helping, there is a finite prefix γ of β such that there is a linearization $\mathsf{lin}(\gamma)$ containing $\mathsf{get}_p()$. Moreover, for every extension λ of γ, there is a linearization $\mathsf{lin}'(\lambda)$ such that $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$. Intuitively, this means that the linearization order of $\mathsf{get}_p()$ in γ is fixed, hence it is univalent at the end of γ. We formally prove this.

Let v be the return value of $\mathsf{get}_p()$ in $\mathsf{lin}(\gamma)$. We claim that $\mathsf{get}_p()$ is v-univalent in the configuration at the end of γ. Let λ be any extension of γ in which $\mathsf{get}_p()$ is completed. Let u be the output value of $\mathsf{get}_p()$ in λ. Since $\mathsf{get}_p()$ is completed in λ, any linearization of λ contains $\mathsf{get}_p()$. As noted above, there is a linearization $\mathsf{lin}'(\lambda)$ of λ such that $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$, which implies that $u = v$. We conclude that $\mathsf{get}_p()$ is v-univalent at the end of γ. We now show that $v \neq \bot$. Observe that $\mathsf{lin}(\gamma)$ must have the form $S' \cdot S'' \cdot \langle \mathsf{get}_p() : \mathsf{return}\, v\rangle \cdot S'''$, for some sequences S'' and S''' of put operations by q. Note that $S' \cdot S'' \cdot S'''$ is a prefix of

S, since q executes its operations in the order they appear in S. Thus, by property P1, it follows that $v \neq \perp$.

Finally, since $\mathsf{get}_p()$ is multivalent at the end of α' and it is univalent on a non-$\perp$ value at the end of γ, there must be a prefix of γ that ends in a configuration C in which $\mathsf{get}_p()$ is multivalent but it is univalent in $q(C)$ on a non-$\perp$ value. Therefore, $\mathcal{A}$ has nontrivial helping. ◀

▶ **Corollary 10.** *A wait-free linearizable implementation of a queue or stack with universal helping has nontrivial helping.*

Proof. For the case of the stack, S is the infinite execution in which some process q performs a sequence of *push* operations with distinct values and S' is any non-empty prefix of S. The case of the queue is defined similarly. ◀

Figure 5 presents a stack implementation that has nontrivial helping but not universal helping, as established by Lemma 14 in the appendix. It augments the wait-free stack of Afek et al. [1] with a helping mechanism, added by lines 01–05 in push and lines 08–14 in pop. Each process $p_i \neq p_n$ that wants to push value x, first checks if p_n's current pop operation is pending (lines 02–03), and if so, tries to help p_n by directly giving x to its current pop operation. If p_i succeeds in updating $H[j]$ in line 05, then it does not access the *items* array. In that case, p_n is not able to update $H[j]$ in line 11, implying that it must take the value in *h_items* that p_i left for it (lines 12–14). If p_n manages to update $H[j]$ in line 11, then no process succeeded in helping it and it proceeds as in Afek et al.'s stack (lines 15–23). Similarly, processes whose *Push* operation fails to help p_n proceed as in Afek et al.'s stack (lines 06–07).

In Appendix A, we prove that the algorithm in Figure 5 is a wait-free linearizable implementation of a stack that has nontrivial helping but not universal helping. Together with Lemma 9, this implies that, for stacks, universal helping is strictly stronger than nontrivial helping.

5 Related Notions

This section compares valency-based helping and universal helping to the definition of helping in [3] and the notion of strong linearizability [9].

5.1 Relation to the help definition of [3]

Helping is formalized in [3] as follows. A linearizable implementation of a concurrent object has helping, which we call here *linearization-based helping*, if there is an execution α with distinct operations op_1 and op_2 by p and q, such that

1. there are linearizations $\mathsf{lin}(\alpha)$ and $\mathsf{lin}'(\alpha)$ such that op_1 precedes op_2 in $\mathsf{lin}(\alpha)$ and op_2 precedes op_1 in $\mathsf{lin}'(\alpha)$, and
2. in every linearization of $\alpha \cdot r$, for some $r \neq p$ (possibly $r = q$), op_1 precedes op_2.

In a sense, valency-based helping and linearization-based helping are incomparable. On the one hand, valency-based helping allows us to distinguish stacks and queues, as queues need nontrivial (valency-based) helping and stacks do not (Theorems 3 and 6), while both stacks and queues necessarily have linearization-based helping [3]. On the other hand, valency-based helping cannot capture helping among enqueues, as they always return true. Nevertheless, enqueues *are* taken into account, since the dequeues in an execution reveal how the helping mechanism determines the order of enqueues.

```
Shared Variables:
    $range$ : Fetch&Add register initialized to 1
    $current$ : read/write register initialized to 1
    $items$ : array $[1, \ldots, ]$ of read/write registers
    $h\_items$ : 2-dimensional array $[1, \ldots, n-1, 1, \ldots, ]$ of read/write registers
    $pend$ : array $[1, \ldots, ]$ of boolean read/write registers initialized to false
    $T$ : array $[1, \ldots, ]$ of Test&Set objects
    $H$ : array $[1, \ldots, ]$ of Compare&Swap objects initialized to $\perp$

Operation Push($x$):
    if $ID \neq n$ then
        $j \leftarrow current$
        if $pend[j]$ then
            $h\_items[ID][j] \leftarrow x$
            if Compare&Swap($H[j], \perp, ID$) then return true
        end if
    end if
    $i =$ Fetch&Add($range, 1$)
    $items[i] \leftarrow x$
    return true
end Push

Operation Pop():
    if $ID = n$ then
        $j \leftarrow current$
        $pend[j] \leftarrow$ true
        if $\neg$Compare&Swap($H[j], \perp, ID$) then
            $current \leftarrow j + 1$
            $k \leftarrow$ get($H[j]$)
            return $h\_items[k][j]$
        end if
    end if
    $t =$ Fetch&Add($range, 0$)
    for $i \leftarrow t$ downto 1 do
        $x \leftarrow items[i]$
        if $x \neq \perp$ then
            if Test&Set($T[i]$) then
                if $ID = n$ then $current \leftarrow j + 1$
                return $x$
            end if
        end if
    end for
    if $ID = n$ then $current \leftarrow j + 1$
    return $\perp$
end Pop
```

Figure 5 A stack implementation without universal helping.

A proof similar to the proof of Lemma 9 shows that universal helping implies linearization-based helping. Consider any implementation (wait-free or nonblocking) of a data type with universal helping. Consider an infinite execution β that starts in an initial configuration, where a process p starts some operation op_1 and stops before the operation is completed, and afterwards a distinct process q completes infinitely many operations (the type does not matter). Since the implementation has universal helping, eventually, the order of op_1 in any linearization is fixed. More precisely, there is a prefix γ of β such that, for every linearization of every extension of γ, the order of op_1 is the same. This implies that, at some point, a step of q made op_1 precede an operation op_2 of q, in every linearization. Thus, the implementation has linearization-based helping.

The other direction is not necessarily true, since one can modify Herlihy's universal construction to get a nonblocking implementation of any data type from Compare&Swap in which each process is helped just once. The resulting implementation has linearization-based helping but not universal helping because universal helping requires that *every* pending operation is eventually linearized, which does not happen once every process in the execution has been helped, since from this point on some operations may be blocked forever.

▶ **Theorem 11.** *For every data type T, every nonblocking or wait-free implementation of T with universal helping has linearization-based helping, while the opposite is not necessarily true.*

5.2 Relation to strong linearizability [9]

Roughly speaking, an implementation of a data type is *strongly linearizable* [3] if once an operation is linearized, its linearization order cannot be changed in the future. More specifically, there is a function L mapping each execution to a linearization, and the function is *prefix-closed*: for every two executions α and β, if α is a prefix of β, then $L(\alpha)$ is a prefix of $L(\beta)$.

In a sense, universal helping can be thought of as a sort of *eventual* strong linearizability. For every execution α, as it is extended, there is eventually an extension α' with a linearization $\mathsf{lin}(\alpha\,\alpha')$ such that for every execution β, if $\alpha\,\alpha'$ is a prefix of β, then there is a linearization $\mathsf{lin}'(\beta)$ with $\mathsf{lin}(\alpha\,\alpha') = \mathsf{lin}'(\alpha\,\alpha')$. We stress that universal helping provides the property that pending operations are linearized eventually, which is not guaranteed by strong linearizability.

The simulation in the proof of Theorem 8 solves consensus because from some point on, all processes agree on a first operation and this agreement cannot be changed as a result of future steps. The following theorem can be proven using a simulation similar to the one in the proof of Theorem 8, with the difference being that each process only needs to complete a single enqueue because the linearization order of that operation does not change in the future.

▶ **Theorem 12.** *Let $\mathcal{B}$ be an n-process strongly-linearizable nonblocking implementation of a queue (stack). Then, n-process consensus can be solved from $\mathcal{B}$.*

The previous theorem shows that, for some data types, strong linearizability for n processes can only be obtained through consensus number n, thus strong linearizability is costly, even if we are looking for nonblocking implementations. However, for stacks, linearizability can be obtained from consensus number 2 as there are wait-free stack implementations from Test&Set [1].

▶ **Corollary 13.** *There is no n-process strongly-linearizable nonblocking implementation of a queue (stack) from primitives with consensus number less than n.*

All previous impossibility results on strongly-linearizable implementations that we are aware of consider only implemenations from consensus-number 1 base objects [7, 10].

6 Discussion

We have considered two ways to formalize helping in implementations of shared objects, one that is based on operation valency and another that is based on possible linearizations. We used these notions to study the kind of helping needed in wait-free implementations of queues and stacks, from Test&Set and stronger primitives. In this work we used an ad-hoc definition of nontrivial helping for queues and stacks, but this notion can be generalized by defining two disjoint sets of outputs values, *trivial* and *nontrivial*, and defining trivial and nontrivial helping accordingly. These notions might facilitate further study of the relations between nonblocking and wait-free implementations.

References

1 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007.

2 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, 1993.

3 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, 2015.

4 Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8_10.

5 Matei David. Wait-free linearizable queue implementation. Master's thesis, Department of Computer Science, University of Toronto, 2004.

6 Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 137–151, 2005. doi:10.1007/11561927_12.

7 Oksana Denysyuk and Philipp Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *Distributed Computing – 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 60–74, 2015. doi:10.1007/978-3-662-48653-5_5.

8 David Eisenstat. A two-enqueuer queue. *CoRR*, abs/0805.0444, 2008. URL: http://arxiv.org/abs/0805.0444.

9 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC'11, pages 373–382, 2011.

10 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *ACM Symposium on Principles of Distributed Computing, PODC'12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 385–394, 2012. doi:10.1145/2332432.2332508.

11 Danny Hendler and Nir Shavit. Operation-valency and the cost of coordination. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC'03, pages 84–91, 2003.

12 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

13 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

14 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, Department of Computer Science, University of Toronto, 2001.

A Stack without Universal Helping

▶ **Lemma 14.** *The algorithm in Figure 5 is a wait-free linearizable implementation of a stack that has nontrivial helping but no universal helping.*

Proof. We first prove that the implementation is linearizable (clearly, it is wait-free). Let α be an execution of the algorithm. Intuitively, we show that there is an execution γ of Afek et al.'s stack implementation (see Figure 2) such that the operations in γ respect the real-time order of the operations in α and the outputs are the same. Thus, α is linearizable since γ is linearizable.

In any execution of the algorithm, a $\mathsf{push}_i(x)$ operation of p_i *matches* the j-th $\mathsf{pop}_n()$ operation of p_n, if p_i successfully updates $H[j]$ during the execution. We call such a pair of operations a *matching*.

Let k be the number of matchings in α. By induction on k, we show that α is linearizable. If $k = 0$, then α is linearizable because it corresponds to some execution of Afek et al.'s implementation. Suppose that the claim holds for $k - 1$. Below we show that it holds for k.

Let $\mathsf{push}_i(x)$ by p_i and $\mathsf{pop}_n()$ by p_n be the k'th matching in α. Note that $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$ are concurrent in α. Moreover, the Compare&Swap in line 05 of $\mathsf{push}_i(x)$ precedes the Compare&Swap in line 11 of $\mathsf{pop}_j()$ (if p_n ever executes it).

We now construct an execution α' that is easier to reason about than α. Let β be the longest prefix of α that does not have the Compare&Swap in line 05 of $\mathsf{push}_i(x)$. Thus, in the configuration at the end of β, p_i is about to perform the Compare&Swap in line 05, and p_n is about to perform the Compare&Swap in line 11.

Let β_i and β_n respectively denote the subsequences of $\alpha - \beta$ containing only the steps of $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$. Let λ be the subsequence of $\alpha - \beta$ obtained by removing the steps of β_i and β_n.

Then, α' is the execution $\beta\, \beta_i\, \beta_n\, \lambda$. Intuitively, in α', the steps of $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$ are placed together.

It can be seen that there is no process that can distinguish between α and α': since neither p_i nor p_n change *items* or *range* in $\mathsf{push}_i(x)$ and $\mathsf{pop}_n()$, the position in the execution when they take the steps in β_i and β_n does not affect other operations. Moreover, α' respects the real-time order in α: if an operation op_1 precedes op_2 in α, op_1 also precedes op_2 in α'. Although there may be concurrent operations in α that are not concurrent in α', this is not a problem for linearizability.[1] Therefore, if α' is linearizable, then α is linearizable too. We now show that it is.

Consider the following execution γ that starts with β and then:

1. p_n executes the Compare&Swap in line 11 (hence sets $H[j]$).
2. p_i executes three consecutive steps, which correspond to lines 05, 06 and 07 (because it cannot set $H[j]$).
3. If $\mathsf{pop}_j()$ (by p_n) is completed in α', p_n completes it in γ (thus it outputs x).
4. If $\mathsf{push}_i(x)$ is completed in α', p_i completes it in γ.
5. λ is appended at the end.

Thus, in γ, p_n is about to take its output from *items*, p_i places x in *items* (at the top of the stack) and p_n takes it from there. The steps of λ, following $\mathsf{push}_i(x)$ and $\mathsf{pop}_j()$, proceed as in α and the only difference is that the Fetch&Add in lines 06 and 15 outputs in γ an integer larger than in α, since p_i adds 1 to *range* in γ in operation $\mathsf{push}_i(x)$.

Also observe that γ respects the real-time order in α'. By induction hypothesis, γ is linearizable, since it has $k - 1$ matchings. Let $\mathsf{lin}(\gamma)$ be a linearization of γ. From the

[1] For example, in α', $\mathsf{push}_i(x)$ precedes any operation starting in λ, however, in α those operations might be concurrent.

properties of γ just described, it follows that $\mathsf{lin}(\gamma)$ is actually a linearization of α' as well, hence a linearization of α. Therefore, the implementation is linearizable.

We now show that the algorithm has nontrivial helping. Starting at the initial configuration, let α be the execution in which p_n completes alone a $\mathsf{push}(1)$ operation and then starts a $\mathsf{pop}()$ operation and stops just before executing the Compare&Swap in line 11. Then, p_1 starts a $\mathsf{push}(2)$ operation and stops just before executing the Compare&Swap in line 05.

Let C be the configuration at the end of α. We claim that the $\mathsf{pop}()$ is multivalent in C. Indeed, let $x \geq 3$. In the extension of α in which first p_2 completes a $\mathsf{push}(x)$ alone and then p_n completes its $\mathsf{pop}()$, the output of the $\mathsf{pop}()$ is x. Also, note that $\mathsf{pop}()$ is 2-univalent in $p_1(C)$ because there is no extension of $p_1(C)$ in which p_n updates $H[1]$ in Line 11, so if it ever returns a value, this must be the value in $h\_items[1][1]$ (which is 2).

Finally, we prove that the implementation has no universal helping. Starting at the initial configuration, let α be the execution in which p_1 starts $\mathsf{pop}()$ and stops before executing the Fetch&Add in line 15. Let β the the infinite extension of α in which p_2 completes alone (infinitely many) push operations with distinct values. If the algorithm would have had universal helping, then there would have been a finite prefix γ of β such that there was a linearization $\mathsf{lin}(\gamma)$ containing pop(), and for every extension λ of γ, there would have been a linearization $\mathsf{lin}'(\lambda)$ such that $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$.

Let γ be such a prefix of β and let λ be the extension of γ in which p_2 completes any pending operation in γ and a $\mathsf{push}(x)$, where x is greater than any value in γ. Let λ' be the extension of λ in which p_1 completes its pop() operation. Observe that p_1's operation outputs x in λ'. Moreover, there is no linearization $\mathsf{lin}'(\lambda')$ of λ' with $\mathsf{lin}(\gamma) = \mathsf{lin}'(\gamma)$ because $\mathsf{push}(x)$ does not appear in γ. Thus, the implementation has no universal helping. ◀

Generic Proofs of Consensus Numbers for Abstract Data Types

Edward Talmage[1] **and Jennifer Welch**[2]

1 **Parasol Laboratory, Texas A&M University, College Station, USA**
etalmage@tamu.edu

2 **Parasol Laboratory, Texas A&M University, College Station, USA**
welch@cse.tamu.edu

Abstract

The power of shared data types to solve consensus in asynchronous wait-free systems is a fundamental question in distributed computing, but is largely considered only for specific data types. We consider general classes of abstract shared data types, and classify types of operations on those data types by the knowledge about past operations that processes can extract from the state of the shared object. We prove upper and lower bounds on the number of processes which can use data types in these classes to solve consensus. Our results generalize the consensus numbers known for a wide variety of specific shared data types, such as compare-and-swap, augmented queues and stacks, registers, and cyclic queues. Further, since the classification is based directly on the semantics of operations, one can use the bounds we present to determine the consensus number of a new data type from its specification.

We show that, using sets of operations which can detect the first change to the shared object state, or even one at a fixed distance from the beginning of the execution, any number of processes can solve consensus. However, if instead of one of the first changes, operations can only detect one of the most recent changes, then fewer processes can solve consensus. In general, if each operation can either change shared state or read it, but not both, then the number of processes which can solve consensus is limited by the number of consecutive recent operations which can be viewed by a single operation. Allowing operations that both change and read the shared state can allow consensus algorithms with more processes, but if the operations can only see one change a fixed number of operations in the past, we upper bound the number of processes which can solve consensus with a small constant.

1998 ACM Subject Classification E.1 Distributed Data Structures

Keywords and phrases Distributed Data Structures, Abstract Data Types, Consensus Numbers, Distributed Computing, Crash Failures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.32

1 Introduction

Determining the power of shared data types to implement other shared data types in an asynchronous crash-prone system is a fundamental question in distributed computing. Pioneering work by Herlihy [7] focused on implementations that are both wait-free, meaning any number of processes can crash, and linearizable (or atomic). As shown in [7], this question is equivalent to determining the *consensus number* of the data types, which is the maximum number of processes for which linearizable shared objects of a data type can be used to solve the consensus problem. If a data type has consensus number n, then in a system with n processes, shared objects of this type can be used to implement shared objects

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 32; pp. 32:1–32:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of any other type. Thus, knowing the consensus number of a data type gives us a good idea of its computational strength.

We wish to provide tools with which it is easy to determine the consensus number of any given data type. So far, most known consensus number results are for specific data types. These are useful, since we know the upper and lower bounds on the strength of many commonly-used objects, but are of no help in determining the consensus number of a new shared data type. Further, even among the known bounds, there are some that seem similar, and even have nearly identical proofs of their bounds, but these piecemeal proofs for each data type give no insight into those relations.

1.1 Summary of Results

We define a general schema for classifying data types, based on their sequential specifications, which we call *sensitivity*. If the information about the shared state which an operation returns can be analyzed to extract the arguments to a particular subsequence of past operations, we say that the data type is sensitive to that subsequence. For example, a register is sensitive to the most recent write, since a read returns the argument to that write. A stack is sensitive to the last *Push* which does not have a matching *Pop*, since a *Pop* will return the argument to that *Push*. We define several such classes in this paper, such as data types sensitive to the kth change to the state, data types sensitive to the kth most recent change, and data types sensitive to the l consecutive recent changes.

We show a number of bounds, both upper and lower, on the number of processes which can use shared objects whose data types are in these different sensitivity classes to solve wait-free consensus. Specifically, we begin by showing that information about the beginning of a history of operations of a shared data type allows processes to solve consensus for any number of processes. This is a natural result, since the ordering of operations on the shared objects allows the algorithm to break symmetry.

An augmented queue, as in [7], using *Enqueue* and *Peek* is such a data type, as *Peek*s can always determine what value was enqueued first, and all processes can decide that value. Other examples include a Compare-and-Swap (CAS) object using a function which stores its argument if the object is empty and returns the contents without changing them, if it is not. Repeated applications of this operation have the effect of storing the argument to the first operation executed and returning it to all subsequent operations. There are data types which are stronger than this, such as with operations which return the entire history of operations on the shared object, but our result shows that that strength is unneeded for consensus.

Next, we consider what happens if a data type has operations which depend on the last operations executed. We show that if a data type has only operations whose return values depend exclusively on one operation at a fixed distance back in history, then that data type can only solve consensus for a small, constant number of processes. A data type whose operations which cannot atomically both read and change the shared state, consensus is only possible for one process. If a data type's operations reveal some number l of consecutive changes to the shared state, then it can solve consensus for l processes.

These data types model the scenario when there is limited memory. If we want to store a queue, but only have enough memory to store k elements, we can throw away older elements, yielding a data type sensitive to recent operations. A cyclical queue has such behavior, and with operations *Enqueue* and *Peek*, where *Peek* returns the kth-most recent argument to *Enqueue*, has consensus number 1. To solve consensus for more processes with a similar data type, we show that knowledge of consecutive past operations is sufficient. If instead of only one recent argument, we can discern a contiguous sequence of them, we can solve consensus for more processes. Using the same cyclical k-queue, if our *Peek* operation is replaced with a *ReadAll* which tells the entire contents of the queue atomically, we show that we can solve consensus for k processes. This parameterized result suggests a fundamental property of the amount of necessary information for solving consensus.

1.2 Related Work

Herlihy[7] first introduced the concepts of consensus numbers and the universality of consensus in asynchronous, wait-free systems. He showed that a consensus object could provide a wait-free and linearizable implementation of any other shared object. Further, he showed that different objects could only solve consensus for certain numbers of processes. This gives a hierarchy of object types, sorted by the maximum number of processes for which they can solve consensus. He also proved consensus numbers for a number of common objects.

Many researchers have worked to understand exactly what level of computational power this represents, and when consensus numbers make sense as a measure of computational power. Jayanti and Toueg [8] and Borowsky, et al. [3] established that consensus numbers of specific data types make sense when multiple objects of the type and R/W registers are used, regardless of the objects' initial states. Bazzi et al. [2] showed that adding registers to a deterministic data type with consensus number greater than 1 does not increase the data type's consensus number. Other work establishes that non-determinism collapses the consensus number hierarchy [9, 10], that consensus is impossible with Byzantine [1], and what happens when multiple shared objects can be accessed atomically [11].

Ruppert [12] provides conditions with which it is possible to determine whether a data type can solve consensus. He considers two generic classes of data types, RMW types and readable types. RMW types have a generic Read-Modify-Write operation which reads the shared state and changes it according to an input function. Readable types have operations which return at least part of the state of the shared object without changing it. He shows that for both of these classes, consensus can be solved among n processes if and only if they can discern which of two groups the first process to act belonged to. This condition, called n-discerning, is defined in terms of each of the classes of data types. This has a similar flavor to our first result below, where seeing what happened first is useful for consensus. We define our conditions more directly as properties of the sequential specification of a shared object and also consider different perspectives on what previous events are visible.

Chordia et al. [5] have lower bounds on the number of processes which can solve consensus using classes of objects with definitions similar to [12]–the duration for which two operation orderings are distinguishable affects the objects' consensus power–using algebraic properties, as we do. These results are not directly comparable to those in [12], since they have different assumptions about the algorithms and exact data returned. [5] also does not provide upper bounds, which we focus on.

In another direction, Chen et al. [4] consider the edge cases of several data types, when operations' return values are not traditionally well-defined. An intuitive example is the effect of a *Dequeue* operation on an empty queue, where it could return $\perp$ or return an arbitrary value, never return a useful value again, or a number of other possibilities. They consider a few different possible failure modes, and show that the consensus numbers of objects are different when they have different behaviors when the object "breaks" in such a case. These results are orthogonal to our paper, as they primarily focus on queues and stacks, and assume that objects break in some permanent way when they hit such an edge case. We assume that there is a legal return value for any operation invocation, and that objects will continue to operate even after they hit such an edge case.

2 Definitions and Model

We consider a shared-memory model of computation, where the programming system provides a set of shared objects, accessible to processes. Each object is linearizable (or atomic) and thus will be modeled as providing operations that occur instantaneously. Each object has an *abstract data type*, which gives the interface by which processes will interact with the object. A data type T provides two things: (1) A set of operations OPS which specify an association of arguments and return values as *operation instances* $OP(arg, ret), OP \in OPS$ and (2) A sequential specification ℓ_T which is a set of all the *legal* sequences of operation instances. We use arg_{OP} and ret_{OP} to denote the sets of possible arguments and return values, respectively, to instances of operation OP. Given any sequence ρ of operation instances, we use $\rho|_{args}$ to denote the sequence of arguments to the instances in ρ.

We assume the following constraints on the set of legal sequences:

- *Prefix Closure*: If a sequence ρ is legal, every prefix of ρ is legal.
- *Completeness*: If a sequence ρ is legal, for every operation OP in the data type and every argument $arg \in arg_{OP}$, there exists a response $ret \in ret_{OP}$ such that $\rho \cdot OP(arg, ret)$ is legal (where $\cdot$ is concatenation).
- *Determinism*: If a sequence $\rho \cdot OP(arg, ret)$ is legal, there is no $ret' \neq ret$ such that $\rho \cdot OP(arg, ret')$ is legal.

We say that two finite legal sequences ρ_1 and ρ_2 of operation instances are *equivalent* (denoted $\rho_1 \equiv \rho_2$) if and only if for every sequence ρ_3, the sequence $\rho_1 \cdot \rho_3$ is legal if and only if $\rho_2 \cdot \rho_3$ is legal.

We classify all operations of a data type into two classes, not necessarily disjoint. Informally, *accessors* return some value about the state of a shared object and *mutators* change the state of the object. An operation may be both an accessor and a mutator, in which case we call it a *mixed* operation. If it is an accessor but not a mutator, we say that it is a *pure* accessor. Similarly, pure mutators are mutators but not accessors. Formally,

▶ **Definition 1.** An operation OP of an abstract data type T is a *mutator* if there is some legal sequence ρ of instances of operations of T and some instance op of OP such that $\rho \not\equiv \rho \cdot op$.

▶ **Definition 2.** An operation OP of an abstract data type T is an *accessor* if there is some legal sequence ρ of instances of operations of T, an instance op of some operation of T such that $\rho \cdot op$ is legal, and an instance aop of OP such that $\rho \cdot aop$ is legal, but $\rho \cdot op \cdot aop$ is not legal.

We consider only data types with non-vacuous sets of operations, which include both a mutator and an accessor (not necessarily distinct). Any shared object which does not have a mutator is a constant which can be replaced by a local copy and any shared object without an accessor is of no use to any party, since they cannot discern the state of the object. We further consider only data types whose operation set has at least one mutator which accepts at least two distinct arguments.

2.1 Sensitivity

We will use the concept of *sensitivity* to classify operations. The sensitivity of a set of operations is a means of tracking which previous operations on a shared object cause a particular instance to return a specific value. Intuitively, an operation which has a return value will usually return a value dependent on some subset of previous operation instances.

For example, a *read* on a register will return the argument to the last previous *write*. On a queue, an instance of *Dequeue* will return the argument of the first *Enqueue* instance which has not already been returned by a *Dequeue*. We categorize operations by which previous instances (first, latest, first not already used, etc.) we can deduce, or "see", based on the return value of an instance of an accessor operation.

▶ **Definition 3.** Let OPS be a subset of the operations of a data type T. Let OPS_M denote the set of all mutators in OPS. Let S be an arbitrary function that, given a finite sequence $\rho \in \ell_T$, returns a subsequence of ρ consisting only of instances of mutators.

OPS is defined to be S*-sensitive* iff there exist an accessor $AOP \in OPS$ and a computable function $decode : ret_{AOP} \rightarrow$ the set of finite sequences over $\bigcup_{MOP \in OPS_M} arg_{MOP}$ such that for all $\rho \in \ell_T$, $arg \in arg_{AOP}$, and $ret \in ret_{AOP}$ with $\rho \cdot AOP(arg, ret) \in \ell_T$, $decode(ret) = S(\rho)|_{args}$.

▶ **Definition 4.** A subset OPS of the operations of a data type T is *strictly* S*-sensitive* if for every $\rho \in \ell_T$, every accessor AOP and every instance $AOP(arg, ret)$ with $\rho \cdot AOP(arg, ret) \in \ell_T$, $ret = S(\rho)|_{args}$. That is, $AOP(arg, ret)$ gives no knowledge about the shared state except for $S(\rho)|_{args}$.

An example, for which we will later show bounds on the consensus number, is k*-front-sensitive* sets of operations:

▶ **Definition 5.** A subset OPS of the operations of a data type T is k*-front-sensitive* for a fixed integer k if OPS is S-sensitive where $S(\rho)$ is the kth mutator instance in ρ for every $\rho \in \ell_T$ consisting of instances of operations in OPS which has at least k mutator instances.

In an augmented queue (as in [7]), the operation set $\{Enqueue, Peek\}$ is k-front-sensitive by this definition, where $k = 1$, S returns the first mutator in a sequence of operation instances, the accessor AOP is $Peek$, and the *decode* function is the identity, since the return value of $Peek$ is the argument to the first $Enqueue$ on the queue. In fact, this operation set is also strictly 1-front-sensitive, since the return value of an instance of $Peek$ is the argument to the single first $Enqueue$.

2.2 Consensus

We are studying the binary *Consensus* problem in an *asynchronous wait-free* model with n processes. In an asynchronous model, processes have no common timing. One process can perform an unbounded number of actions before another process performs a single action. A wait-free model allows for up to $n-1$ processes to fail by *crashing*. A process which crashes ceases to perform any further actions. Processes may fail at any time and give no indication that they have crashed. Processes which do not crash are said to be *correct*. Any algorithm running in this model must be able to continue despite all other processes crashing, while it cannot in a bounded amount of time distinguish between a crashed process and a slow process. Thus, any algorithm in this model must never require a process to wait for any other process to complete an action or reach a certain state.

We say that an *execution* of an algorithm using a shared data type is a sequence of operation instances, each labeled with a specific process and shared object. The projection of an execution onto a single object must be a *legal* operation sequence, by the sequential specification of the data type.

The consensus problem is defined as follows: Every process has an initial *input* value $v \in \{0, 1\}$. After that, if it is correct, it will *decide* a value $d \in \{0, 1\}$. Once a process decides a value, it cannot change that decision. Further, all correct processes must satisfy three conditions:

- Termination: All correct processes eventually decide some value
- Agreement: All correct processes decide the same value d
- Validity: All correct processes decide a value which was some process' input

An abstract data type T can *implement* consensus if there is an algorithm in the given model which uses objects of T (plus registers) to solve consensus. The *consensus number* of an abstract data type is the largest number of processes n for which there exists an algorithm to implement consensus among n processes using objects of that data type. If there is no such largest number, we say the data type has consensus number ∞.

We use valency proofs, as in [7], to show upper bounds on the number of processes for which an abstract data type can solve consensus. The following lemma was implicit in [7] and made explicit in [12]. We will use this to make proofs of upper bounds on consensus numbers cleaner.

To state the lemma, we recall the concepts of *valency* and *critical configurations*. A *configuration* represents the local states of all processes and the states of all shared objects. When a process p_i executes a step of a consensus algorithm, it causes the system to proceed from one configuration C to another, which we call a *child configuration*, and denote by $p_i(C)$. A configuration is *bivalent* if it is possible, starting from that configuration, for the algorithm to cause all processes to decide 0 and also possible for it to cause all processes to decide 1. A configuration is *univalent* if from that configuration, the algorithm will necessarily cause processes to always reach the same decision value. If this value is 0, the configuration is *0-valent* and if it is 1, the configuration is *1-valent*. A configuration is *critical* if it is bivalent, but all its child configurations are univalent.

▶ **Lemma 6.** *Every critical configuration has child configurations with different valencies which are reached by different processes acting on the same shared object, which cannot be a register.*

We also restate the following lemma based on Fischer et al. [6].

▶ **Lemma 7.** *A consensus algorithm always has an initial bivalent configuration and must have a reachable critical configuration in every execution.*

Note that we do not require that the set of sensitive operations is the entire set of operations supported by the shared object(s) in the system. There may be other operations. These extra operations do not detract from the ability of a sensitive set of operations to solve consensus, since an algorithm may just choose not to use any other operations. This means that our proofs of the ability to solve consensus are powerful. Impossibility proofs do not get this extra strength, as a clever combination of operations which are not sensitive in a particular way may allow stronger algorithms.

3 k-Front-Sensitive Data Types

We begin by proving a result that generalizes the consensus number of augmented queues. We observe that if all processes can determine which among them was the first to modify a shared object, then they can solve consensus by all deciding that first process' input. For, example, in an augmented queue, any number of processes can solve consensus by each enqueuing their input value, then using peek to determine which enqueue was first [7].

More generally, processes do not need to know which mutator was first, as long as they can all determine, for some fixed integer k, the argument of the kth mutator executed on the shared object. Thus, we have the following general theorem, which applies to either a mutator and pure accessor or to a mixed operation. An example (for $k = 1$) is an augmented queue, where *Peek* returns the first argument ever passed to an *Enqueue*, requiring no decoding. Another similar example is a Compare-And-Swap operation which places a value into a shared register in an initial state and leaves any other value it finds in the object, leaving the argument of the first operation instance still in the shared object, and thus decodable at each subsequent operation. For any k, a mixed operation which stores a value and returns the entire history of past changes, satisfies the definition, since the first argument is always visible to later operations.

▶ **Theorem 8.** *The consensus number of a data type containing a k-front-sensitive subset of operations is ∞.*

We give a generic algorithm (Algorithm 1) which we can instantiate for any k-front-sensitive set of operations (which has a mutator with at least two possible distinct arguments) to solve consensus among any number of processes and prove its correctness as a consensus algorithm. The mutator and accessor in the algorithm are not necessarily distinct operations.

Algorithm 1 Consensus algorithm for a data type with a k-front-sensitive subset of operations, OPS, using a mutator OP and accessor AOP, in OPS

```
1: for $i = 1$ to $k$ do
2:     $OP(input)$
3: end for
4: $result \leftarrow AOP(arg)$                    ▷ Arbitrary argument $arg$
5: $val \leftarrow decode(result)$
6: $decide(val)$
```

Proof. We must show that this algorithm satisfies the three properties of a consensus algorithm.

- *Termination*: Each process performs a finite number of operations, never waiting for another process. Thus, even in a wait-free system, where any number of other processes may have crashed, all running processes will terminate in a finite length of time.
- *Validity*: By the definition of sensitivity, the decision value at each process will be an argument to a past mutator, and only processes' input values are passed as inputs to mutators on the shared object. Thus, each decision value is some process' input value, and is valid.
- *Agreement*: $decode(result)$ will return the argument to the kth mutator instance at all processes. Since each process completes k mutators before it invokes AOP, there are guaranteed to be at least k mutators preceding the instance of AOP in line 4. Thus, each process decides the same value.

No part of the algorithm or proof is constrained by the number of participating processes, which means that this algorithm solves consensus for any number of processes using a k-front-sensitive data object, so the consensus number of any shared object with a k-front-sensitive set of operations is ∞. ◀

4 Consensus with End-Sensitive Data Types

While data types which “remember” which mutator went first, or kth as above, are intuitively very useful for consensus, other data types can also solve consensus, though not necessarily for an arbitrary number of processes. As a motivating example, consider the difference in semantics and consensus numbers between stacks and queues, shown in [7]. Both store elements given them in an ordered fashion, and the basic version of each has consensus number 2. However, adding extra power to a queue in the form of a *peek* operation gives it consensus number ∞, while adding a similar operation *top* to stacks does not give them any extra power.

If we view the difference between an augmented queue and an augmented stack in terms of sensitivity, *Enqueue* and *Peek* on a queue are front-sensitive, while *Push* and *Top* on a stack are end-sensitive. That is, queues see what operation was first, while stacks see which was latest. When processes cannot tell how far in the algorithm other processes have gotten, though, due to asynchrony, knowing what operation was latest is not helpful for consensus, as another mutator could finish after some process decides, and that other process will see a different last value. We explore generalizations of this problem and what power still remains in end-sensitive data types.

Unfortunately, the picture for data types with end-sensitive operations sets is more complex than that for front-sensitive types. Here, we have variations depending on exactly which part of the end of the previous history is visible or partly visible to an accessor. It is also important that shared objects have a pure accessor, or some other means of maintaining the state of the object, or else every operation will change what future operations see, making it difficult or impossible to come to a consensus.

We begin with a symmetric definition to that in Section 3, but for recent operations instead of initial, and show that it is not useful for consensus. We then show that certain subclasses, which are sensitive to more than one past operation, have higher consensus numbers.

▶ **Definition 9.** A subset OPS of the operations of a data type T is *k-end-sensitive* for a fixed integer k if OPS is S-sensitive where $S(\rho)$ is the kth-last mutator instance in ρ for every $\rho \in \ell_T$ consisting entirely of instances of operations in OPS and containing at least k mutator instances, and $S(\rho)$ is a null operation instance $\perp(\perp, \perp)$, if there are not at least k mutator instances in ρ.

This definition does not lead to as simple a result as that for front-sensitive sets of operations. As we will show, there is no algorithm for solving consensus for n processes with an arbitrary k-end-sensitive set of operations, for $n > 1$. We will give a number of more fine-grained definitions, showing that different subsets of the class of k-end-sensitive operation sets range in power from consensus number 1 to consensus number ∞.

Consider a set of operations which is S-sensitive, where for all ρ, $S(\rho)$ is the entire sequence of mutator instances in ρ. This set of operations is both k-end-sensitive and k-front-sensitive, for $k = 1$. By the result from Section 3, we know that such a set of operations has consensus number ∞. A similar result holds for any k for which an operation set is k-front-sensitive. Thus, in this section, we will only consider operation sets which are not k-front-sensitive for any k and consider only the strength and limitations of end-sensitivity.

4.1 k-End-Sensitive Types

Unlike front-sensitive data types, if a set of operations is strictly k-end-sensitive, for some fixed k, the data type does not have infinite consensus number. This is a result of the fact that the kth-last mutator is a constantly moving target, as processes execute more mutators. As we will show, in an asynchronous system, if there are more than one or three processes in the system (depending on the types of operations in the set), operations can be scheduled such that the "moving target" is always obscured for some processes, so they cannot distinguish which process took a step first after a critical configuration, which prevents them from safely deciding any value. We formalize this in the following theorems.

▶ **Theorem 10.** *For $k > 2$, any data type with a strictly k-end-sensitive operation set consisting only of pure accessors and pure mutators has consensus number 1.*

Proof. Suppose we have a consensus algorithm A for at least 2 processes, p_0 and p_1, using such an operation set. Consider a critical configuration C of an execution of algorithm A, as per Lemmas 6, 7. If p_0 is about to execute a pure accessor, p_1 will not be able to distinguish C from the child configuration $p_0(C)$ when running alone, by the definition of a pure accessor. Thus, it will decide the same value in the executions where it runs from either of those states, which contradicts the fact that they have different valencies. If p_1's next operation is a pure accessor, a similar argument holds.

Thus, both processes' next operations from configuration C must be mutators. Assume without loss of generality that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent. Then the states $C_0 = p_1(p_0(C))$ and $C_1 = p_0(p_1(C))$ are likewise 0-valent and 1-valent, respectively.

We construct a pair of executions, extending C_0 and C_1, in which at least one process cannot learn which configuration it is executing from. By the Termination condition for consensus algorithms, at least one process must decide in a finite number of steps, and since the two executions return the same values to the first process to decide, it will decide the same value after $p_1(p_0(C))$ as after $p_0(p_1(C))$, despite those configurations having different valencies. This is a contradiction to the supposed correctness of A, showing that no such algorithm can exist.

We construct the first execution, from C_0, as follows. Assuming for the moment that both processes continue to execute mutators (we will discuss what happens when they don't, below), let p_0 run alone until it is ready to execute another mutator. Then pause p_0 and let p_1 run alone until it is also ready to execute a mutator, and pause it. Let p_0 run alone again until it has completed $k-2$ mutators and is ready to execute another. Next, allow p_1 to run until has executed one mutator, and is prepared to execute a second. We then continue to repeat this sequence, allowing p_0 to run alone again for $k-2$ mutators, then p_1 for one, etc.

The second execution is constructed identically from C_1 except that after C_1, p_0 first runs until it has executed $k-3$ mutators and is ready to execute another, then p_1 executes a mutator. After that, the processes alternate as in the first execution, with p_0 executing $k-2$ mutators and p_1 executing one.

We know that each process, running alone from C_0 (or C_1), must execute at least $k-2$ mutators to be able to see what mutator was first after C, since we have a strictly k-end-sensitive set of operations, which means that any correct algorithm must execute at least that many mutators, since it must be able to distinguish $p_0(C)$ from $p_1(C)$. The way we construct the executions, though, we interleave the operation instances in such a way that each process sees only its own operation instances, and cannot distinguish these executions from running alone from C_0 (or C_1). It is an interesting feature of this construction that we do not force any processes to crash. In fact, we need both processes to continue running to ensure that they successfully hide their own operations from each other.

If we denote any mutator by m and any accessor by a, with subscripts to indicate the process to which the operations belong and superscripts for repetition (in the style of regular expressions), we can represent these two execution fragments, restricted to the shared object operated on in configuration C, as follows:

$$m_0 \cdot m_1 \cdot \cdot a_0^* \cdot a_1^* \cdot (m_0 \cdot a_0^*)^{k-2} \cdot (m_1 \cdot a_1^*) \cdot (m_0 \cdot a_0^*)^{k-2} \cdots$$

$$m_1 \cdot m_0 \cdot \cdot a_1^* \cdot a_0^* \cdot (m_0 \cdot a_0^*)^{k-3} \cdot (m_1 \cdot a_1^*) \cdot (m_0 \cdot a_0^*)^{k-2} \cdots$$

Since the return value of each accessor is determined by the kth most recent mutator, all operations are pure, and operations are deterministic, we can see that corresponding accessor instances will return the same value in the two executions. Thus, neither process can distinguish the two executions. This is true despite the possibility of operations on other shared objects. To discern the two runs, each process must determine which process executed an operation first after C, and that can only be determined by operations on this shared object. Thus, as long as the return values to operations on this object are the same, since the algorithm is deterministic, the processes will continue to invoke the same operations in the two runs, and will be unable to distinguish the two executions.

This interleaving of operation instances works as long as both processes continue to invoke mutators. Each process must decide after a finite time, though, so they cannot continue to invoke mutators indefinitely. When a process ceases to invoke mutators, we can no longer schedule operations as before to continue hiding its past operations. There are two possible cases for which process(es) finish their mutators first in the two executions.

First, one process (WLOG p_0) may execute its last mutator before the other does, in both executions. When p_0 executes its last mutator in each execution, let it continue to run alone until it decides. Since configuration C, it has only seen its own mutators, and since the data type is strictly k-end-sensitive and no more mutators are executed, will continue to see only its own past mutators in both executions. Thus, the two executions are identical for p_0 and it will decide the same value in both, contradicting their differing valencies.

Second, it may be that in one execution, p_0 executes its last mutator before p_1 does and in the other, p_1 executes its last mutator before p_0. Each process will follow the same progression of local states in both executions, so this case can only arise when p_0's last mutator in the first execution is the last in a block of $k-2$ mutators it runs by itself, and thus first in such a block in the second execution. In the first execution, after p_0 executes its last mutator, let it run alone, as in the first case. In the second execution, after p_1 executes its last mutator, pause it, and allow p_0 to run alone, executing its last mutator and continuing until it decides. By the same argument as case 1, p_0 decides the same value in both executions, contradicting the fact that they have the same valency.

Thus, the assumed consensus algorithm cannot actually exist. ◀

If mixed operations are allowed, the above proof does not hold, as a mixed operation immediately after C will potentially have a different return value than it would in a different execution where there is an intervening mutator. We can show the following:

▶ **Theorem 11.** *For $k > 2$, any data type with an operation set which is strictly k-end-sensitive has consensus number at most 3.*

The proof of this theorem is almost identical to the previous, and is therefore omitted. The primary difference, which yields a higher bound, is that the first two processes which execute operations in a critical configuration crash immediately after those operations, since they may have seen different shared states depending on their order. The other two (assumed) processes can continue in a manner similar to the above, hiding their executions from each other, and not satisfying the univalency of the configurations, yielding a contradiction.

4.2 1- and 2-End-Sensitive Types

The bounds in the previous section require $k > 2$, so we here explore what bounds hold when $k \leq 2$. We continue to consider strictly k-end-sensitive operations; we will consider operation sets with knowledge of additional operations (that is, with larger sensitive sequences $S(\rho)$) later.

We first consider the case $k = 1$, which implies that accessor operations can see the last previous mutator. If all operations are pure mutators or accessors, then it is intuitive that consensus would not be possible, since we could schedule operations such that each process only saw its own mutators. We show that this is, in fact, the case. This generalizes the bound that registers can only solve consensus for one process. If mixed operations are allowed, then a process can obtain some information about other operations, which we will show is enough to solve consensus for two processes, but no more. We know that this bound of 2 is tight, that is, no lower bound can be proved for the entire class, since $Test\&Set$, for example, is sensitive to only the last previous mutator and has consensus number 2 [7].

▶ **Theorem 12.** *Any data type with a strictly 1-end-sensitive operation set with no mixed operations has consensus number 1.*

▶ **Theorem 13.** *Any data type with a strictly 1-end-sensitive operation set has consensus number at most 2.*

The proofs for these theorems are standard bivalency proofs, and can be found in the full version of the paper: Technical Report 2015-11-1 at `http://www.cse.tamu.edu/research/tr`.

Next, we consider $k = 2$. If the sensitive set of operations includes a pure accessor, we show that we can solve consensus for 2 processes. Here, unlike our other results, the presence or absence of a mixed operation does not seem to affect the strength for consensus. Instead, it is important to have a pure accessor, which can see the 2nd-last mutator without changing it, which makes it practical for both processes to see the same value.

Data types without a pure accessor seem to have less power than consensus, since it is impossible to check the shared state without changing it. This makes it very difficult for processes to avoid confusing each other. A similar argument to that for Theorem 11 provides an upper-bound of $n \leq 3$ for this data type. We conjecture that it is lower($n = 1$), but do not yet have the tools to prove this formally.

For now, an upper bound on the consensus number of 2-end-sensitive operation types is an open question, but we conjecture that it will be 2, or perhaps 3 with mixed operations as for k-end-sensitive types with $k > 2$, above.

▶ **Theorem 14.** *For $k = 2$, a data type containing a k-end-sensitive set of operation types which includes a pure accessor has consensus number at least 2, using Algorithm 2.*

The proof of Theorem 14 is left to the full version.

4.3 Knowledge of Consecutive Operations

Operation sets which only allow a process to learn about one past operation are generally limited to solving consensus for at most a small constant number of processors. We now show that knowledge about several consecutive recent operations allows more processes to solve consensus. In effect, we are enlarging the moving target we discussed before. We will show that this does, in fact, allow consensus algorithms on more processes, as many as the size of the target, or the number of consecutive operations we can decode. We will then show that when we know the last mutator instances that have happened, the bound is tight.

Algorithm 2 Consensus Algorithm for 2 processes using 2-end-sensitive set of operations using mutator OP and pure accessor AOP

```
$OP(input)$
$val \leftarrow AOP()$
if $decode(val) = \bot$ then
    $decide(input)$
else
    $decide(decode(val))$
end if
```

This is interesting because the consensus number is not affected by how old the visible operations are, as long as they are at a consistent distance. That is, if we always know a window of history that is a certain fixed number of operations old (no matter what that number is), we can use it to solve consensus. Also interesting is the fact that the bound is parameterized. While knowing a single element of history can solve consensus for a constant number of processes, if we know l consecutive mutators in the history, we can solve consensus for l processes for any natural number l. Thus, knowing more consecutive elements always increases the consensus number.

We could use this to create a family of data types which solve consensus for an arbitrary number of processes, with a direct cost trade-off. If we maintain a rolling cache of several consecutive mutators, we trade off the size of the cache we maintain against the number of processes which can solve consensus. If we only need consensus for a few processes, we know we only need to maintain a small cache. If we have the available capacity to maintain a large cache, we can solve consensus for a large number of processes.

We begin by defining the sensitivity of these large-target operation sets, and giving a consensus algorithm for them. In effect, the algorithm watches for the target to fill up, and as long as it is not full, can determine which process was first. Since we can only see instances as long as the target "window" does not overflow, this gives the maximum number of processes which can use this algorithm to solve consensus. We later show this number is tight, if there are no mixed operations.

▶ **Definition 15.** A subset OPS of the operations of a data type T is *l-consecutive-k-end-sensitive* for fixed integers l and k if OPS is S-sensitive where for every $\rho \in \ell_T$, $S(\rho)$ is the sequence of l consecutive mutator instances in ρ, the last of which is the kth-last mutator instance in ρ. If there are not that many mutator instances in ρ, the missing ones are replaced by $\bot(\bot, \bot)$ in $S(\rho)$.

▶ **Theorem 16.** *Any data type with an l-consecutive-k-end-sensitive set of operations has consensus number at least l, using Algorithm 3.*

We will show that this is the maximum possible number of processes for which we can give an algorithm which solves consensus using any l-consecutive-k-end-sensitive operations set. We do this by considering a special case of that class, l-consecutive-0-end-sensitive with only pure operations, and showing that the bound is tight for it. As with most end sensitive classes, a set of operations which satisfies the definition of l-consecutive-k-end-sensitive may also be sensitive to more, earlier operations, and thus have a higher consensus number. We will show a particular example of such an operation set, to show that there is more work to be done to classify end-sensitive data types.

Theorem 17 below shows an upper bound on the consensus number of strictly l-consecutive-0-end-sensitive operation sets. That is, operation sets in which accessors can learn exactly the

Algorithm 3 Consensus algorithms for l processes using an l-consecutive-k-end-sensitive operation set. (A) Using mutator OP and pure accessor AOP. (B) Using mixed operation BOP.

(A)

```
for $x = 1$ to $k$ do
    $OP(input)$
    $vals[1..l] \leftarrow decode(AOP())$
    let $m = \arg\min_{n \in 1..l}\{vals[n] \neq \bot\}$
    if $m$ exists then
        $decide(vals[m])$
    end if
end for
```

(B)

```
for $x = 1$ to $k$ do
    $vals[1..l] \leftarrow decode(BOP(input))$
    let $m = \arg\min_{r \in 1..l}\{vals[r] \neq \bot\}$
    if $m$ exists then
        $decide(vals[m])$
    end if
end for
$decide(input)$
```

last l mutators. To achieve this bound, we need to restrict ourselves to operation sets which have no mixed accessor/mutator operations. This is a strong restriction, but we will give an example showing that a mutator which also returns even a small amount of information about the state of the shared object can increase the consensus number of an operation set. The proof of Theorem 17 is given in the full version of the paper.

▶ **Theorem 17.** *Any data type with a strictly l-consecutive-0-end-sensitive set of operations which has no mixed accessor/mutators has consensus number at most l.*

There are sets of operations which are strictly l-consecutive-0-end-sensitive, but have a mixed operation which returns information about the state of the object. We here give an example such set. Specifically, the mixed operation returns a (limited) count of the number of preceding mutators. Even this small amount of extra information is enough to increase the consensus power of a set of operations.

Consider an l-element shared cyclic queue with operations $Enq_l(x)$ and $ReadAll()$. $Enq_l(x)$ is a mixed accessor/mutator which adds x to the tail of the queue, discarding the head element if there are more than l elements in the queue, and returning the number of Enq_l operations which have previously been executed, up to l. If more than l Enq_l operations have been previously executed, the return value will continue to be l. $ReadAll()$ is a pure accessor which returns the entire contents of the l-element queue. This is clearly a strictly l-consecutive-0-end-sensitive set of operations, since the return values of $ReadAll()$ and Enq_l depend on the last l $Enq_l(x)$ calls, but only the last l are visible to each instance of one of these. We show that it has consensus number at least $l + 1$ by giving Algorithm 4.

The intuition for this algorithm is that all processes but one will be able to see which process was first. The variable *state* will tell how many previous Enq_l instances processes have executed. If this is less than k, all previous Enq_ls are visible, and the process can return the input of the first. If there have been k previous Enq_ls, then we cannot see the first, but we know that there are at most $l + 1$ processes and each executed only one Enq_l, so the one process whose Enq_l we cannot see must have been first, and we decide that process' input.

This algorithm shows that mixed operations can give extra strength for consensus, beyond sensitivity, which is difficult to quantify. In general, mixed operations can not only give different return values based on the state of the shared object, but can alter the way they modify the object's state based on its previous state. This allows them to preserve any non-empty state, which means that it can keep a record of which process first modified

Algorithm 4 Algorithm for each process i to solve consensus for $l+1$ processes using a l-element cyclic queue with *Enq*$_l$ and *ReadAll*

```
$Write_i(input)$                                        ▷ In a shared SWMR register
$state \leftarrow Enq_l(i)$
$l\_history \leftarrow (ReadAll())$
if There are $state$ values preceding $i$ in $l\_history$ then
    decide oldest element in $l\_history$
else
    $j \leftarrow$ processor id not appearing in $l\_history$
    decide $Read_j()$                                   ▷ Value from $p_j$'s SWMR register
end if
```

the state, giving a front-sensitive data type, which can solve consensus for any number of processes. For example, a *Read-Modify-Write* operation can exhibit this behavior.

5 Conclusion

We have defined a number of classes of operations for shared objects, and explored their power for solving consensus. First, we generalized, with an intuitive result, the common understanding that knowing what process acted on a shared object first allows a consensus algorithm for any number of processes. We then considered what might be possible if only knowledge about recent operations, instead of initial operations, is available.

Here, because the set of recent operations is constantly changing, we must be more precise about what knowledge is available. If operations cannot both change and view the shared state atomically, then the number of processes which can solve consensus is given by the number of consecutive changes a process can view atomically. Further, these do not need to be the most recent changes, as long as processes know how old the data they receive is.

If operations can atomically view and change the shared state, then they generally have the potential for more computational power. We show in a few cases that if an operation set has a mixed operation, then it can solve consensus for one more process than a similarly-sensitive operation set without a mixed operation. Unfortunately, mixed operations may be more expensive to implement than pure accessors or mutators, which would lead to a trade-off between computational power and operation cost.

We point out that the quantity of information learned in a single atomic step is a dominating factor in a data type's consensus power. This appears strongly in types sensitive to consecutive past mutators and weakly in the marginally greater power of mixed operations.

We summarize our results in Table 1. We have results for front-sensitive sets of operations and several subclasses of end-sensitive operation sets. Several of these classes have different consensus numbers if we allow mixed accessor/mutator operations or only allow pure accessors and pure mutators, so we separate those results. Note also that all upper bounds further assume a data type with a strictly sensitive set of operations.

In future work, we wish to fill missing entries in the above table. In addition, we wish to further explore conditions on the knowledge of the execution which operations can extract to classify more operations. More generally, the idea of exploring how information travels through the execution history of a shared object, affecting the return values of different subsequent operations in different ways, is fascinating. As currently defined, sensitivity cannot classify all possible operation sets, so an exploration of classifying and providing generic results for other shared data types is of interest.

Table 1 Summary of Upper and Lower Bounds on Consensus Numbers.

Operation Set			Lower Bounds		Upper Bounds	
			Pure	Mixed	Pure	Mixed
Front-sensitive			∞		-	
End-Sensitive	k-end:	$k > 2$	1	?	1	3
		$k = 1$	1	2	1	?
		$k = 2$	2	?	?	3
	l-consecutive-k-end		l		l $(k = 0)$	?

Another direction is to consider trade-offs between the implementation costs of shared operations and their consensus numbers. It would be interesting to develop a metric which balances an operation's cost with its computational strength. Finding minima of such a metric would be an interesting result, potentially showing the optimal cost for solving consensus for any given number of processes.

Acknowledgments. This work was supported in part by NSF grants 0964696, 1526725. We would also like to thank the anonymous reviewers for their helpful comments.

References

1 Paul C. Attie. Wait-free byzantine consensus. *Inf. Process. Lett.*, 83(4):221–227, 2002. doi:10.1016/S0020-0190(01)00334-9.

2 Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. On the use of registers in achieving wait-free consensus. *Distributed Computing*, 10(3):117–127, 1997. doi:10.1007/s004460050029.

3 Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'94, pages 363–372, New York, NY, USA, 1994. ACM. doi:10.1145/197917.198126.

4 Wei Chen, Guangda Hu, and Jialin Zhang. On the power of breakable objects. *Theor. Comput. Sci.*, 503:89–108, 2013. doi:10.1016/j.tcs.2013.05.036.

5 Sagar Chordia, Sriram K. Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Asynchronous resilient linearizability. In Yehuda Afek, editor, *Distributed Computing – 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2013. doi:10.1007/978-3-642-41527-2_12.

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

7 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

8 Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG'92, Haifa, Israel, November 2-4, 1992, Proceedings*, volume 647 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 1992. doi:10.1007/3-540-56188-9_5.

9 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. doi:10.1137/S0097539798335766.

10 Ophir Rachman. Anomalies in the wait-free hierarchy. In Gerard Tel and Paul M. B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG'94, Terschelling, The Netherlands, September 29 – October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 156–163. Springer, 1994. doi:10.1007/BFb0020431.

11 Eric Ruppert. Consensus numbers of multi-objects. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC'98, Puerto Vallarta, Mexico, June 28 – July 2, 1998*, pages 211–217. ACM, 1998. doi:10.1145/277697.277736.

12 Eric Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4):1156–1168, 2000. doi:10.1137/S0097539797329439.

Non Trivial Computations in Anonymous Dynamic Networks*

Giuseppe Di Luna[1] and Roberto Baldoni[2]

1 Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti, Università degli Studi di Roma La Sapienza, Via Ariosto, 25, I-00185 Rome, Italy
diluna@dis.uniroma1.it

1 Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti, Università degli Studi di Roma La Sapienza, Via Ariosto, 25, I-00185 Rome, Italy
baldoni@dis.uniroma1.it

Abstract

In this paper we consider a static set of anonymous processes, i.e., they do not have distinguished IDs, that communicate with neighbors using a local broadcast primitive. The communication graph changes at each computational round with the restriction of being always connected, i.e., the network topology guarantees 1-interval connectivity. In such setting non trivial computations, i.e., answering to a predicate like *"there exists at least one process with initial input a?"*, are impossible. In a recent work, it has been conjectured that the impossibility holds even if a distinguished leader process is available within the computation. In this paper we prove that the conjecture is false. We show this result by implementing a deterministic leader-based terminating counting algorithm. In order to build our counting algorithm we first develop a counting technique that is time optimal on a family of dynamic graphs where each process has a fixed distance h from the leader and such distance does not change along rounds. Using this technique we build an algorithm that counts in anonymous 1-interval connected networks.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Distributed System, Anonymous Networks, Dynamic Networks

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.33

1 Introduction

Highly dynamic distributed systems are attracting a lot of interest from the relevant research community [13, 7]. These models are well suited to study the new challenges introduced by distributed systems where there is an immanent dynamicity given by the presence of mobile devices, unstable communication links and environmental constraints. A critical element in such future distributed systems is the *anonymity* of the devices; the uniqueness of a process ID is not guaranteed due to operational limit (e.g., in highly dynamic networks maintaining unique IDs may be infeasible due to mobility and failure among processes [22]) or to maintaining user's privacy (e.g., where users may not wish to disclose information about their behavior [11]).

In this paper we consider a static set of anonymous process $|V|$, this set of processes is connected by a dynamic communication graph that is governed by a fictional omniscient

* This work has been partially supported by the TENACE project (MIUR-PRIN 20103P34XC).

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 33; pp. 33:1–33:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

entity *the adversary* who has the power to change at each round the graph. The adversary is able to read the local memory of each process in order to deploy the worst possible communication graph to challenge the computation. The only restriction imposed to the adversary is that the graph has to be connected at each round. This corresponds to the 1-interval connectivity model proposed in [12].

We focus on the problem of counting the number of processes in the system which is one of the fundamental problems of distributed computing [12, 17, 10, 21, 2]. The system model employed in this paper also assumes each process communicates with its neighbors using a local broadcast primitive. Under this model, it has been proved that the presence of a leader process is *necessary* in order to compute non trivial tasks [19]. In the case of leader absence, the adversary could indeed perpetually generate an anonymous ring graph of unknown size, and it is well known that in such graph non-trivial computation are impossible [1, 25]. Let us remark that a leader is actually present in many realistic settings, such as a base station in a mobile network, a gateway in a sensors network etc. Additionally, the computability in the model where all processes are anonymous but a leader has been widely investigated in the static network case and in population protocols [24, 26, 8, 4]. Furthermore, having a leader can be sometimes simpler than ensuring an unique ID for each process. From a formal point of view, it has been proved when processes communicate using broadcast, assuming the existence of a leader is strictly weaker than assuming unique IDs [19].

Anonymity and the adversarial dynamic graph make the system model very challenging for performing non-trivial computation. More specifically, it has been conjectured in [19, 18] that, in such model, the presence of a leader is not sufficient to compute, non trivial tasks such as counting. The main result of this paper is to show that the conjecture is false, and that a distinguished leader process is necessary and sufficient to do deterministic non trivial computations on anonymous 1-interval connected dynamic networks with broadcast. This is shown by introducing a deterministic terminating counting algorithm, namely EXT.

The paper introduces one by one the main sub-algorithms forming EXT, namely OPT, VCD and InstanceCount. As second result presented in this paper, we show that OPT is a time optimal counting algorithm for graphs in $\mathcal{G}(\mathrm{PD})_2$, a specific subset of interval connected dynamic graphs where each process has a fixed distance h from the leader with $h \leq 2$ and such distance is fixed across rounds. We showed in [14] that counting on $\mathcal{G}(\mathrm{PD})_2$ is function of the network size even if there is unlimited bandwidth and a constant dynamic diameter w.r.t. $|V|$. Thus OPT shows that the bound introduced in [14] for counting in $\mathcal{G}(\mathrm{PD})_2$ is tight.

Outline

Section 4 presents an optimal algorithm for graphs belonging to $\mathcal{G}(\mathrm{PD})_2$. Section 5 illustrates the basic structure of the counting algorithm, EXT, for 1-interval connected networks. EXT has two main components: VCD and InstanceCount which are introduced in Section 6 and Section 7 respectively. Finally, we prove that the conjecture presented in [19, 18] is false in Section 8. Due to lack of space, some of the proofs can be found in the full version of the paper[1].

[1] `https://midlab.diag.uniroma1.it/publications.php`

2 Related Work

The question concerning what can be computed on top of static anonymous networks, has been pioneered by Angluin in [1] and it has been the further investigated in many papers [25, 26, 6, 5]. In a static anonymous network with broadcast, the presence of a leader is enough to have a terminating counting algorithm as shown in [18].

Considering dynamic non anonymous networks, counting has been studied under several dynamicity models. In [3], dynamicity corresponds to processes churn where processes leave and join the system. In [17, 23] dynamicity is governed by a random adversary to model peer-to-peer networks. Finally considering the dynamicity model employed in this paper (worst-case adversary), in [12], a counting algorithm for 1-interval connectivity has been proposed. Other results related to counting can be found in [22] where a model similar to 1-interval connected is considered. In the context of possibly disconnected adversarial network, counting has been studied in [20]. The approaches followed by the latter works are not suitable in the model proposed by this paper, they use the asymmetry introduced by IDs.

Counting in anonymous dynamic networks: In [10], the authors propose a gossip-based protocol to compute aggregation function. The network graph considered by [10] is governed by a fair random adversary, moreover the proposed approach converges to the actual count without having a terminating condition. A similar model and strategy is also used by [9]. The first work investigating the problem of terminating counting in an anonymous network with worst-case adversary and a leader node is [18]. They show that when a process is able to send a different messages to each neighbors, the presence of a leader is enough to have a terminating naming algorithm. For the broadcast case, under the assumption of a fixed known upper bound on the maximum process degree, they provided an algorithm that computes an upper bound on the network size. Building on this result, [15] proposes an exact counting algorithm under the same assumption. Finally, [16] provides a counting algorithm for 1-interval connected networks considering each process is equipped with a local degree detector, i.e. an oracle able to predict the degree of the process before exchanging messages. Other works [12, 21] have investigated leader-less randomized approaches to obtain approximated counting algorithms. We are interested in study how anonymity impacts the computational power of 1-interval connected networks with broadcast, for this reason we assume that processes do not have access to a source of randomness, e.g. they cannot break symmetry by using coin tosses.

3 Model of the computation

We consider a synchronous distributed system composed by a finite static set of processes V. Processes in V are *anonymous*, they initially have no identifiers and execute a deterministic *round-based* computation. Processes communicate through a communication network which is *dynamic*. We assume at each round r the network is stable and represented by a graph $G_r = (V, E(r))$ where $E(r) \subseteq V \times V$ is the set of bidirectional links at round r connecting processes in V.

▶ **Definition 1.** A *dynamic graph* $G = \{G_0, G_1, \dots, G_r, \dots\}$ is an infinite sequence of graphs one at each round r of the computation.

A dynamic graph is 1-interval connected, if, and only if, $G \in \mathcal{G}(\text{1-IC})$, if $\forall G_r \in G$ we have that G_r is connected. The neighborhood of a process v at round r is denoted by

$N(v,r) = \{v' : (v',v) \in E(r)\}$. We say that v has *degree* d at round r iff $|N(v,r)| = d$. Given a round r we denote with $p_{v,v'}$ a path on G_r between v and v'. Moreover we denote as $P_r(v',v)$, the set of all paths between v, v' on graph G_r. The distance $d_r(v',v)$ is the minimum length among the lengths of the paths in $P_r(v',v)$, the length of the path is defined as the number of edges. We consider the computation proceed by exchanging messages through synchronous rounds.

Every round is divided in two phases: (i) *send* where processes send the messages for the current round, (ii) *receive* where processes elaborates received messages and prepare those that will be sent in the next round. Processes can communicate with its neighbors through an *anonymous broadcast* primitive. Such primitive ensures that a message m sent by process v_i at the beginning of a certain round r will be delivered to all its neighbors during round r. A process v floods message m by broadcasting it for each round. If process receives a flooded message m then it starts the flooding of m. The flood of m terminates when it has been received by all processes. We say that a network has dynamic diameter D if for any v and any round r the flood of a message that starts at round r from process v terminates by at most round $r + D$. Intuitively the dynamic diameter is the maximum time needed to disseminate messages to all processes in the network.

Leader-based computation and worst case adversary: We assume the selection of a topology graph at round r is done by an omniscient adversary that may choose at each step the worst configuration to challenge a counting algorithm. Due to the impossibility result shown in [18], we assume any counting algorithm that works over a dynamic graph has a leader process v_l starting with a different unique state w.r.t. all the other processes.

► **Definition 2.** Given a dynamic network G with $|V|$ processes, a distributed algorithm $\mathcal{A}$ solves the counting on G if it exists a round r at which the leader outputs $|V|$ and terminates.

Persistent distance dynamic graphs: Let us characterize dynamic graphs according to the distances among a process v and the leader v_l.

► **Definition 3.** (Persistent Distance over G) Consider a dynamic graph G. The distance between v and v_l over G, denoted $D(v,v_l) = d$, is defined as follow: $D(v,v_l) = d$ iff $\forall r, d_r(v,v_l) = d$.

Let us now introduce a set of dynamic graphs based on the distance between the leader and the processes of a graph.

► **Definition 4** (Persistent Distance set)**.** A graph G belongs to Persistent Distance set, denoted $\mathcal{G}(\mathrm{PD})$, iff $\forall v \in G$, $\exists d \in \mathbb{N}^+$ such that $D(v,v_l) = d$

Graphs in $\mathcal{G}(\mathrm{PD})_2$

Among the dynamic graphs belonging to $\mathcal{G}(\mathrm{PD})$ we can further consider the set of graphs, denoted $\mathcal{G}(\mathrm{PD})_h$, whose processes have maximum distance h from the leader with $1 < h \leq |V|$. Thus, given a graph in $\mathcal{G}(\mathrm{PD})_h$ we can partition its processes in h sets, $\{V_0, V_1, \ldots, V_h\}$, according to their distance from the leader. In Figure 1 there is an example of $\mathcal{G}(\mathrm{PD})_2$ graph. The depicted dynamic graph has dynamic diameter $D = 4$, if process v_0 starts a flood at round 0 this flood will reach process v_3 at round 3. The task of the leader process v_l is to count processes in V_2. Let us notice that if a process knows $|N(v,r) \cap V_1|$ before the receive phase of round r then counting in $\mathcal{G}(\mathrm{PD})_2$ needs $O(1)$ rounds, the algorithm is trivial each

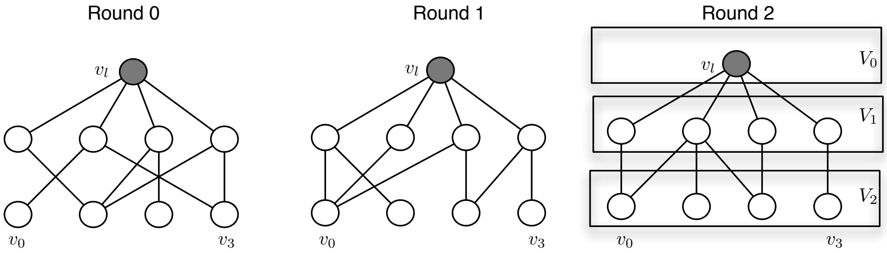


Figure 1 An example of a graph belonging to $\mathcal{G}(\mathrm{PD})_2$ along three rounds.

process in V_2 sends a message $\frac{1}{|N(v,r)\cap V_1|}$ to processes in V_1. A process in V_1 collects these messages and send their sum to the leader. Also if IDs are present counting requires $O(1)$ rounds, in 2 rounds the leader collects the IDs of all processes. It is interesting to notice that if $|N(v,r)|$ is known only when a process receives messages from its neighbors then the time for counting become $\Omega(\log |V|)$ rounds, see Th.2 of [14].

4 An asymptotically optimal algorithm for $\mathcal{G}(\mathrm{PD})_2$

OPT initially starts a *get_distance* phase. At the end of this phase each process is aware of its distance from the leader. In $\mathcal{G}(\mathrm{PD})_2$ this phase takes one round and it works as follow: Each process knows if it is the leader or not. This information is broadcast by each process (including the leader) to its neighbors at the beginning of round 0. Thus, at the end of round 0, each process knows if it belongs either to V_1 or to V_2.

Non-leader process behavior

Starting from round 1, a process broadcasts its distance from the leader (i.e., 1 or 2) and each process v in V_2 builds its *degree history* $v.H(r)$ with $r \geq 0$ where $v.H(r)$ is an ordered list containing the number of neighbors of v belonging to V_1 at rounds $[0,\dots,r-1]$. Thus $v.H(r) = [\perp, |N(v,1)\cap V_1|,\dots,|N(v,r-1)\cap V_1|]$.

Starting from round $r > 0$, each $v \in V_2$ broadcasts $v.H(r)$. These histories are collected by each process $v' \in V_1$ and sent to the leader at the beginning of round $r+1$.

Leader behavior

Starting from the beginning of round $r \geq 2$ the leader receives degree histories from each process in V_1. The leader merges histories in a multiset denoted $v_l.M(r)$. Let us remark that $v_l.M(r)$ may contain the same history multiple times.

Data structure: The leader uses $v_l.M(r)$ to build a tree data structure T whose aim is to obtain $|V_2|$. For each distinct history $[A] \in v_l.M(r)$ the leader creates a node $t \in T$ with label $[A]$ and two variables $< m_{[A]}, n_{[A]} >$. $m_{[A]}$ denotes the number of histories $[A]$ in $v_l.M(r)$ and $n_{[A]}$ is the number of processes in V_2 that have sent $[A]$. Following the information flow, at round 2, $v_l.M(2)$ will be formed by a single history $[\perp]$ with multiplicity $m_{[\perp]}$. The leader creates the root of T with label $[\perp]$, value $m_{[\perp]}$, and $n_{[\perp]} =?$ (where ? means unknown value). It is important to remark that m values are directly computed from $v_l.M(r)$ while n values are set by the leader at a round $r' \geq r$ through a counting rule that will be explained later.

The leader final target is to compute $n_{[\perp]}$ which corresponds to the number of processes in V_2.
At round $r+2$ if the leader receives a history $h = [\perp, x_0, \ldots, x_{r-2}, x_{r-1}]$ and $n_{[\perp,x_0,\ldots,x_{r-2}]} =?$, then it creates a node in $t \in T$ with label h and value m_h, this node is a child of the node with label $[\perp, x_0, \ldots, x_{r-2}]$. Otherwise the leader ignores h. It is straightforward to see that the following equations hold:

$$\begin{cases} m_{[\perp,x_0,\ldots,x_{r-2},x_{r-1}]} = \sum_{i=1}^{|V_1|} i \cdot n_{[\perp,x_0,\ldots,x_{r-2},x_{r-1},i]} \\ n_{[\perp,x_0,\ldots,x_{r-2},x_{r-1}]} = \sum_{i=1}^{|V_1|} n_{[\perp,x_0,\ldots,x_{r-2},x_{r-1},i]} \end{cases} \tag{1}$$

where $i \cdot n_{[\perp,x_0,\ldots,x_{r-2},x_{r-1},i]}$ means that the leader received i copies of history $[\perp, x_0, \ldots, x_{r-2}, x_{r-1}]$, one for each process in V_2 that at round $r+1$ had history $[\perp, x_0, \ldots, x_{r-2}, x_{r-1}, i]$.

Counting Rule: When in T there is a non-leaf node with label $[\perp, x_0, \ldots, x_{r-2}, x_{r-1}, x_r]$ such that the leader knows the number of processes (i.e., $n_{[A]}$), for each of its children but one (i.e., $n_{[\perp,x_0,\ldots,x_{r-1},x_r,j]} =?$). Then the leader computes $n_{[\perp,x_0,\ldots,x_{r-1},x_r,j]}$ using $m_{[\perp,x_0,\ldots,x_{r-2},x_{r-1},x_r]} = \sum_{i=1}^{|V_1|} i \cdot n_{[\perp,x_0,\ldots,x_{r-2},x_{r-1},x_r,i]}$.

When the leader knows the values n for each of the children of a non leaf-node t, it sums the children values and sets the n_t (see the second equation of Eq. 1).

Due to the fact that the number of processes is finite, eventually there will be a non-leaf node in T with only one child (a leaf). Thanks to the counting rule, the n variables of the child and of the father will be set. This will start a recursive procedure that will eventually set $n_{[\perp]}$ terminating the counting.

In Figure 2 is depicted an example run of the algorithm. In the full version the detailed pseudocode for T is provided.

Correctness proof

► **Lemma 5.** *Let us consider the algorithm OPT. Eventually v_l sets a value for $n_{[\perp]}$ and this value is $|V_2|$.*

Proof. We first prove that eventually we reach a round in which the counting rule can be applied for any leaf of T. Let us consider the subtree of T rooted in the node with label $[A]$, if there is only one child then the counting rule can be applied and $n_{[A]}$ can be computed. Thus let us suppose that $[A]$ has at least two children with labels $[A, x], [A, x']$ with $x \neq x'$. We have that $n_{[A,x]} \geq 1$ and $n_{[A,x']} \geq 1$ because there must be at least one sending process for each degree-history. Considering that $n_{[A]} = \sum_{j=1}^{k} n_{[A,j]}$, it follows that $n_{[A,x]} \leq n_{[A]} - 1$. Iterating this reasoning we have that when the height of the subtree rooted in $[A]$ is greater than $n_{[A]} - 1$, then each leaf has no sibling: when there is a single process sending a certain degree history H, in the next round there will be only one degree history with H as suffix. As a consequence, after at most $n_{[A]}$ rounds, we may apply the counting rule for any leaf of the subtree rooted in $[A]$.

Now we prove by induction that: for each node $v \in T$ if $n_v \neq ?$, then n_v is equal to the number of processes in V_2 that had degree history equal to v at a given round.

Base case, leaf without siblings: Let $v_1 : [x_0, \ldots, x_{r+1}]$ be a leaf without siblings and $v_0 : [x_0, \ldots, x_r]$ its father. v_l sets, according to the counting rule, $n_{v_0} = n_{v_1} = \frac{m_{v_0}}{x_{r+1}}$. From Eq 1 we have $n_{v_0} = n_{v_1}$ which is equal to the number of processes in V_2 that had degree history $[x_0, \ldots, x_r]$.

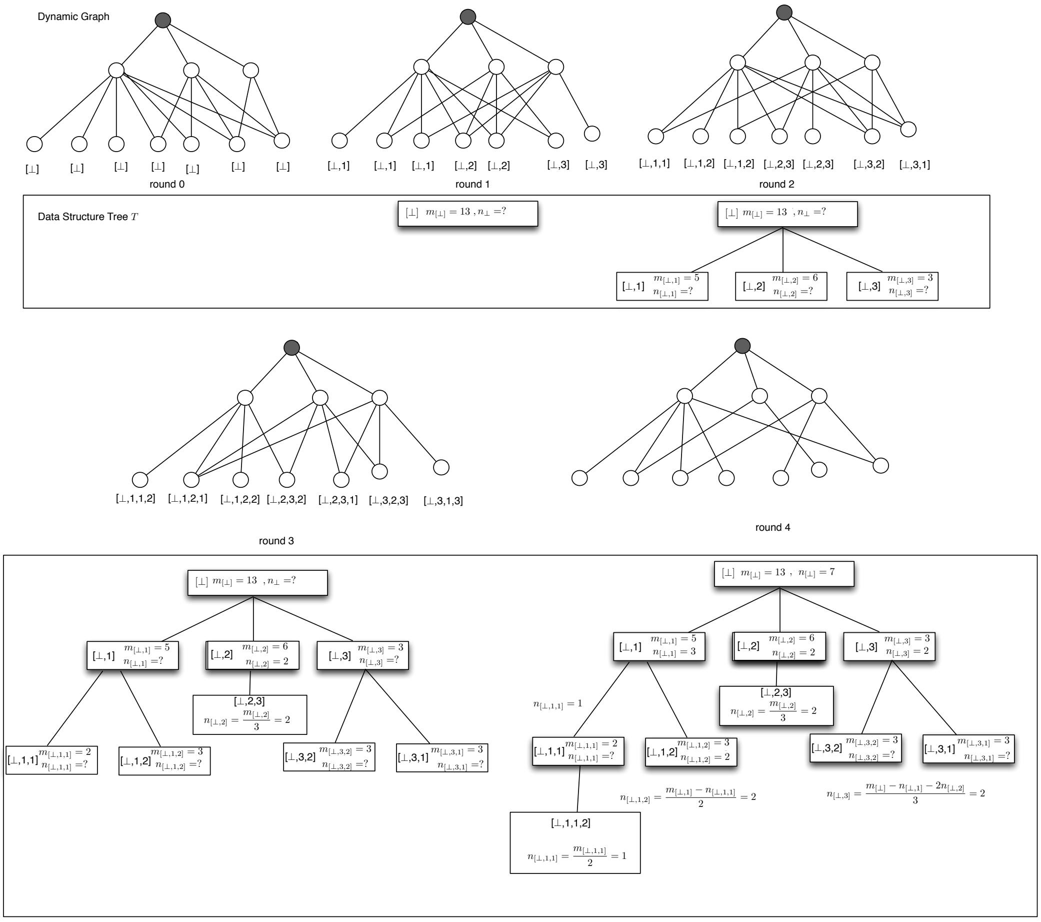

Dynamic Graph
[⊥] [⊥] [⊥] [⊥] [⊥] [⊥] [⊥]
round 0
[⊥,1] [⊥,1] [⊥,1] [⊥,2] [⊥,2] [⊥,3] [⊥,3]
round 1
[⊥,1,1] [⊥,1,2] [⊥,1,2] [⊥,2,3] [⊥,2,3] [⊥,3,2] [⊥,3,1]
round 2
Data Structure Tree T
$[\perp]\ m_{[\perp]} = 13\ , n_\perp = ?$
$[\perp,1]\ m_{[\perp,1]} = 5\ n_{[\perp,1]} = ?$
$[\perp,2]\ m_{[\perp,2]} = 6\ n_{[\perp,2]} = ?$
$[\perp,3]\ m_{[\perp,3]} = 3\ n_{[\perp,3]} = ?$
[⊥,1,1,2] [⊥,1,2,1] [⊥,1,2,2] [⊥,2,3,2] [⊥,2,3,1] [⊥,3,2,3] [⊥,3,1,3]
round 3
round 4
$[\perp]\ m_{[\perp]} = 13\ , n_\perp = ?$
$[\perp,1]\ m_{[\perp,1]} = 5\ n_{[\perp,1]} = ?$
$[\perp,2]\ m_{[\perp,2]} = 6\ n_{[\perp,2]} = 2$
$[\perp,3]\ m_{[\perp,3]} = 3\ n_{[\perp,3]} = ?$
$[\perp,2,3]\ n_{[\perp,2]} = \frac{m_{[\perp,2]}}{3} = 2$
$[\perp,1,1]\ m_{[\perp,1,1]} = 2\ n_{[\perp,1,1]} = ?$
$[\perp,1,2]\ m_{[\perp,1,2]} = 3\ n_{[\perp,1,2]} = ?$
$[\perp,3,2]\ m_{[\perp,3,2]} = 3\ n_{[\perp,3,2]} = ?$
$[\perp,3,1]\ m_{[\perp,3,1]} = 3\ n_{[\perp,3,1]} = ?$
$[\perp]\ m_{[\perp]} = 13\ , n_{[\perp]} = 7$
$[\perp,1]\ m_{[\perp,1]} = 5\ n_{[\perp,1]} = 3$
$[\perp,2]\ m_{[\perp,2]} = 6\ n_{[\perp,2]} = 2$
$[\perp,3]\ m_{[\perp,3]} = 3\ n_{[\perp,3]} = 2$
$[\perp,2,3]\ n_{[\perp,2]} = \frac{m_{[\perp,2]}}{3} = 2$
$n_{[\perp,1,1]} = 1$
$[\perp,1,1]\ m_{[\perp,1,1]} = 2\ n_{[\perp,1,1]} = ?$
$[\perp,1,2]\ m_{[\perp,1,2]} = 3\ n_{[\perp,1,2]} = 2$
$[\perp,3,2]\ m_{[\perp,3,2]} = 3\ n_{[\perp,3,2]} = ?$
$[\perp,3,1]\ m_{[\perp,3,1]} = 3\ n_{[\perp,3,1]} = ?$
$n_{[\perp,1,2]} = \frac{m_{[\perp,1]} - n_{[\perp,1,1]}}{2} = 2$
$n_{[\perp,3]} = \frac{m_{[\perp]} - n_{[\perp,1]} - 2n_{[\perp,2]}}{3} = 2$
$[\perp,1,1,2]\ n_{[\perp,1,1]} = \frac{m_{[\perp,1,1]}}{2} = 1$

Figure 2 A run of OPT algorithm.

Inductive case: Let us consider $v_0 : [x_0, \ldots, x_r]$ and the set of its children C_{v_0} with $|C_{v_0}| > 1$. Let introduce a set X_{v_0} formed by the children for which n is known and set, formally: $X_{v_0} : \{x \in C_{v_0} | n_x \neq ?\}$. If $\exists! v_1 : [x_0, \ldots, x_{r+1}] \in C_{v_0} \setminus X_{v_0}$, the leader sets (according to the counting rule) $n_{v_1} = \frac{m_{v_0} - \sum_{\forall [x_0,\ldots,x_k] \in X_{v_0}} (x_k \cdot n_{[x_0,\ldots,x_k]})}{x_{r+1}}$. By inductive hypothesis we have $\forall x \in X_{v_1}$, n_x is equal to the number of processes in V_2 with degree history equal to x. Due to Eq. 1, we have both n_{v_1} and n_{v_0} will be set to the correct value.

From the previous arguments we have that after at most $|V_2|$ rounds all the leaves of $[\perp]$ have no siblings, thus the counting rule will be applied recursively until the value $n_{[\perp]}$ is set to $|V_2|$. ◀

▶ **Theorem 6.** *Let G be a dynamic graph of size $|V|$ belonging to $\mathcal{G}(PD)_2$. A run of OPT on G terminates in at most $\lceil log_2 |V| \rceil + 3$ rounds.*

Proof. Let consider the algorithm OPT. The latter counts processes in V_2, since the number of processes in V_1 is immediately known by v_l at round 0, thus let us suppose that we are in the worst case i.e., $|V_2| = \mathcal{O}(|V|)$. Let us consider the tree T, given a node $[A]$ the maximum height of the subtree rooted in $[A]$ is a function $h_{max}(n_{[A]})$. We have that h_{max} is non decreasing, $h_{max}(n_{[A]} - 1) \leq h_{max}(n_{[A]})$: let us consider the worst scheduling that the adversary uses with $n_{[A]} - 1$ processes in order to obtain the maximum height. It easy to show that the same scheduling can be created with $n_{[A]}$ processes, the adversary will simply force two processes to follow the behavior of a single process in the old schedule. Let us restrict to the case when $[A]$ has only two children: $[A, x], [A, x']$, for the counting rule $h_{max}(n_{[A]}) = min(h_{max}(n_{[A,x]}), h_{max}(n_{[A,x']}))+1$. Considering the second equation of Eq. 1, $h_{max}(n_{[A]})$ can be rewritten as follows: $h_{max}(n_{[A]}) = 1 + min(h_{max}(\frac{n_{[A]}}{2} - \delta), h_{max}(\frac{n_{[A]}}{2} + \delta)) \leq 1 + min(h_{max}(\frac{n_{[A]}}{2}), h_{max}(\frac{n_{[A]}}{2}))$ with $\delta \in [0, \frac{n_{[A]}}{2}]$. Thus, the optimal height can be reached by having $n_{[A,x]} = n_{[A,x']} = \frac{n_{[A]}}{2}$. Let us notice that when $[A]$ has more than two children, the maximum height of the subtree rooted in $[A]$ cannot be greater than the one obtained when $[A]$ has two children. Iterating this reasoning, in the worst case T is a balanced tree with degree at most 2 for each non leaf node and with exactly $|V|$ leaves. The height of this tree is $\lceil \log_2(|V|) \rceil$. Each level of T corresponds to one round of OPT, this completes the proof. ◀

A $\Omega(\log |V|)$ bound on $\mathcal{G}(\text{PD})_2$ has been shown in [14]. Therefore we have that OPT is asymptotically optimal.

5 High level view of $\mathcal{G}$(1-IC) counting algorithm

[19] and [18] conjectured: *It is impossible to compute (even with a leader) the predicate $Na \geq 1$, that is "exists an a in the input", in general anonymous unknown dynamic networks with broadcast.* In order to show that the conjecture is false we present a terminating counting algorithm, namely EXT, on $\mathcal{G}$(1-IC); this obviously implies the possibility to answer to any existence predicate confuting the conjecture.

Let us introduce the underlying structure we use to build EXT. The first conceptual step is to extend OTP to obtain a counting algorithm on $\mathcal{G}(\text{PD})_h$. We denote this extended algorithm OPT_h. As a second step, we consider networks in $\mathcal{G}$(1-IC). In such networks, at each round, processes can change their distance from the leader in $[1, V - 1]$. When a process changes distance we say that the process *"moved"*.

5.1 Counting in $\mathcal{G}(\text{PD})_h$: OPT_h Algorithm

As OPT, OPT_h begins with a *get_distance* phase over $\mathcal{G}(\text{PD})_h$ where each process v obtains its distance from the leader, $v.distance$. This is done by using a simple flood and convergecast algorithm. After this phase the counting begin.

Each non leader process v keeps a degree history, where each element is the number of processes in $N(v, r)$ whose distance from v_l is $v.distance - 1$. Moreover v updates a multiset $v.M(r)$ that contains messages received by neighbors at distance $v.distance + 1$. The degree history and the multiset are broadcast at each round.

From an high level point of view the algorithm of v_l works as follow: the leader first computes the number of processes in V_1. Then by using messages sent by process in V_1, let MS_1 be this multiset[2], it executes OPT to count the processes in V_2. By counting processes in V_2 it also obtains the multiset MS_2 of messages sent by these processes. At this point, the leader simulates, using MS_2, an execution of OPT counting processes in V_3. Iterating this procedure till processes at distance h we obtain the final count.

[2] Due to anonymity multiple messages from different processes may be undistinguishable.

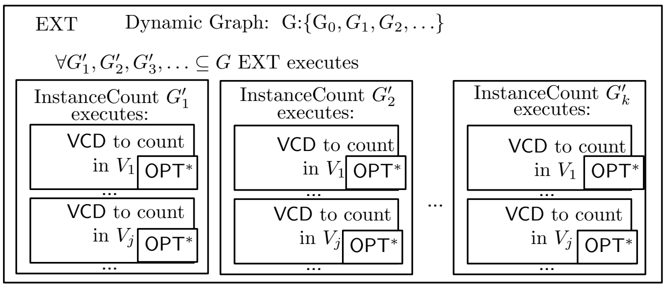

EXT
Dynamic Graph: G:{$G_0, G_1, G_2, \ldots$}
$\forall G'_1, G'_2, G'_3, \ldots \subseteq G$ EXT executes
InstanceCount G'_1 executes:
VCD to count in V_1
OPT*
VCD to count in V_j
OPT*
InstanceCount G'_2 executes:
VCD to count in V_1
OPT*
VCD to count in V_j
OPT*
...
InstanceCount G'_k executes:
VCD to count in V_1
OPT*
VCD to count in V_j
OPT*

Figure 3 Counting algorithm EXT and the relationship among its subalgorithms. Algorithms VCD and OPT* are explained in Section 6; InstanceCount in Section 7. Finally, EXT is presented in Section 8.

Let us remark that OPT_h is an asymptotically time optimal algorithm for graphs in $\mathcal{G}(\text{PD})_h$. A more detailed explanation of OPT_h, with pseudo-code and formal proofs can be found in the full version.

5.2 Using $\mathcal{G}(\text{PD})_h$ to count in $\mathcal{G}$(1-IC)

Let us introduce the notion of temporal subgraph G' of G:

▶ **Definition 7.** (Temporal Subgraph) Given a dynamic graph G, a dynamic graph G' is a temporal subgraph of G ($G' \subseteq G$) if and only if $G' : [G_{i_1}, G_{i_2}, \ldots]$ is an ordered subsequence of $G : [G_0, G_1, G_2, \ldots]$.

We can show that in each $G \in \mathcal{G}$(1-IC) there exists a temporal subgraph G' that belongs to $\mathcal{G}(\text{PD})_h$:

▶ **Lemma 8.** *Let us consider a dynamic graph $G : [G_0, G_1, G_2, \ldots] \in \mathcal{G}$(1-IC). There exists $h \in \mathbb{N}^+$ and $\exists G' \subseteq G$ such that G' is infinite and $G' \in \mathcal{G}(PD)_h$.*

Now, let us define a counting algorithm InstanceCount. Such algorithm works on $G \in \mathcal{G}$(1-IC) and it has two properties: (P1) it terminates giving the correct count on instance $G' \in \mathcal{G}(\text{PD})_h$; (P2) it does not give an incorrect count on $G' \notin \mathcal{G}(\text{PD})_h$. Thus, if $G' \notin \mathcal{G}(\text{PD})_h$ it can terminate giving either a correct count of the network or a special invalid value, i.e. INVCNT. The strategy of EXT is to run a different instance of InstanceCount on each temporal subgraph of G. Due to properties (P1) and (P2), EXT terminates correctly when an instance of InstanceCount outputs a valid count value. For the property (P1) and for Lemma 8, one instance of InstanceCount outputs a valid count value. Consequently, EXT is a correct terminating counting algorithm.

InstanceCount counts as if the network is in $\mathcal{G}(\text{PD})_h$. Therefore, the leader first counts processes in V_1, then processes in V_2 and so on. This is done until v_l counts processes of a set V_h such that no set V_{h+1} exists. The tricky part is to detect if the counting algorithm is operating on a network in $\mathcal{G}(\text{PD})_h$. In the affirmative, the count done with such strategy will be correct. The procedure that counts each set V_j is a special algorithm, namely VCD. VCD allows to detect if the count obtained for V_j is correct, returning the count value, or if it is not possible to count V_j because some process moved during the counting, returning NOCOUNT. The VCD algorithm is explained in the next Section.

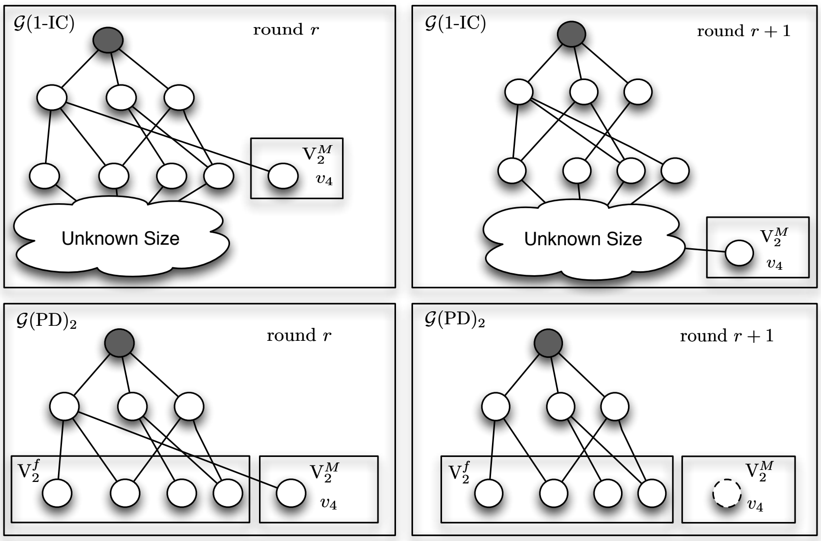


Figure 4 In general $\mathcal{G}$(1-IC) a subset V_2^M of processes in V_2 may move changing the distance from the leader and invalidating the correct count of processes in V_2. Network size is unknown therefore messages from process v_4 need an unknown number of rounds to reach the leader. We are interested in an algorithm that detects this using information from processes in $V_2 \setminus V_2^M$. This is equivalent to solve the problem on a networks in $\mathcal{G}(\mathrm{PD})_2$ where the subset V_2^M stops sending messages after a certain round. In the example process v_4 halts at round $r + 1$.

6 Valid Count Detection Algorithm (VCD)

Let us considering a network where processes in V_1 do not change distance from the leader along rounds. Remaining processes, including a proper subset V_2^M of processes in V_2 at round 0, may change distance along rounds. We wish to build an algorithm that solves this problem: *if no processes in V_2^M move during the counting, then the algorithm outputs the correct count of processes at distance* 2 *at round* 0*. Otherwise, the algorithm outputs either the correct count or a special invalid value.* Unfortunately, in case processes change their distance from the leader, OPT might fail outputting a wrong count.

One strategy to build such algorithm could be to first use OPT then, after OPT termination, to start a waiting phase in order to receive messages from processes that could have moved. Sadly, this simple OPT-based strategy does not work. If v_l outputs the final count at round r, the message from a process that moved could arrive at round $r + 1$, invaliding the count. Thus, v_l should wait for some time before outputting the count but this time cannot be bounded as the size of the network is unknown. From this point of view, if a process changes distance across rounds in a network of unknown size it is like if this process halts and it does not send anymore messages. We denote this problem as Valid Count Detection.

Valid Count Detection Problem (VCDP)

Let us consider a graph in $\mathcal{G}(\mathrm{PD})_2$ where processes in V_2 may halt at some point. We say that v_i *halts at round* r if it has send messages for any round $r'' < r$, and it does not send messages for any $r' \geq r$. We assume that processes halt from round $r \geq 1$; that is they send at least one message before their departure. Now we introduce the **VCDP** problem on $\mathcal{G}(\mathrm{PD})_2$:

▶ **Problem 9** (VCDP). *Given two run R, R_{NC} such that: in the run R no process halts; in the run R_{NC} there are processes that halt. An algorithm solves the Valid Count Detection Problem if at some round r it outputs a value and terminates. The output could be either a special value NOCOUNT or a number $C = |V_2|$. On run R the output value has to be C. On run R_{NC} it could be either C or NOCOUNT.*

The VCD Algorithm to solve VCDP

When identifiers are present a simple broadcast algorithm solves **VCDP** in $\mathcal{G}(\mathrm{PD})_2$. In our model we solve it by using an extension OPT, denoted as OPT*. When processes halt OPT* has a peculiar *"overestimation"* property (see Lemma 11).

Algorithm OPT*

The algorithm OPT* differs from OPT in:

- Its output is considered not valid if we have one of the following: (i) the value n computed for some node of the tree is not in $\mathbb{N}^+$; (ii) if some of the Equations 1 are violated, i.e. $m_{[\bot,x_0,\ldots,x_{r-2},x_{r-1}]} \gtrless \sum_{i=1}^{|V_1|} i \cdot n_{[\bot,x_0,\ldots,x_{r-2},x_{r-1},i]}$; (iii) if at round $r+2$ there exists a node in T with label $[\bot, x_0, \ldots, x_{r-2}, x_{r-1}, x_r]$ and at round $r+3$ does not exists a node with label $[\bot, x_0, \ldots, x_{r-2}, x_{r-1}, x_r, *]$.
- Its counting rule is a restricted version of the OPT counting rule. Specifically: when in T there is a non-leaf node with label $[\bot, x_0, \ldots, x_{r-2}, x_{r-1}, x_r]$ such that *it has only one child* $[\bot, x_0, \ldots, x_{r-1}, x_r, j]$ the leader computes $n_{[\bot,x_0,\ldots,x_{r-1},x_r,j]}$ using:

  $$m_{[\bot,x_0,\ldots,x_{r-2},x_{r-1},x_r]} = j \cdot n_{[\bot,x_0,\ldots,x_{r-2},x_{r-1},x_r,j]} \,.$$

  When the leader knows the values n for each of the children of a non leaf-node t, it sums the children values and sets the n_t (see the second equation of Eq. 1).

Algorithm OPT* has the following properties:

▶ **Lemma 10.** *Let R be a run produced by OPT*. R terminates in $\mathcal{O}(|V_2|)$ rounds.*

▶ **Lemma 11.** *Let R be a run produced by OPT* that starts at round 0 and $|V_2^f|$ be the number of non-halted processes in V_2 at the end of the execution of OPT*. If at some round $r > 0$ processes in V_2 halt, then if the output C of OPT* is valid we have $C > |V_2^f|$.*

Informally the previous Lemma says that, if there are halted processes, the output of OPT* is always an overestimate on the number of non-halted processes. The following lemma states that if no process halts then the output is the number of processes.

▶ **Lemma 12.** *Let R be a run produced by OPT* that starts at round 0. If no process in V_2 halts during the run, then the output of OPT* is valid and it is the correct count of processes in V_2.*

Algorithm VCD

The algorithm executes sequentially k runs of OPT* starting from round 0, for some $k > |V_2|$. The leader compares the output of these runs: if they are all equal and valid, then VCD outputs the count obtained by the first run of OPT*. Otherwise VCD outputs NOCOUNT. The value k is computed by counting the edges connecting processes in V_1 with processes

```
1: $M(-1) = []$
2: $H(-1) = [\bot]$
3: $distance = -1$
4:
5: procedure SENDING_PHASE
6:     send($Message :< distance, M(r), H(r) >$)
7:
8: procedure RCV_PHASE($MultiSet\ MS$)
9:     if $distance == -1 \wedge \exists m \in MS \mid m.distance \neq -1 \wedge m.distance == r$ then
10:        distance=m.distance+1
11:    if $r == distance$ then
12:        for all $m \in MS \mid m.distance == -1$ do
13:            m.distance=distance+1
14:    if $distance \neq -1$ then
15:        if $r > distance \wedge \exists m \in MS \mid m.distance \notin \{distance-1, distance, distance+1\}$ then
16:            $M(r+1) = M(r).append($INVCNT$)$
17:        $H(r+1) = H(r).append(count\_distance\_neighbors(MS, distance-1))$
18:        $M(r+1) = M(r).append(get\_messages\_from\_distance(MS, distance+1))$
```

Figure 5 InstanceCount for $\mathcal{G}$(1−IC): pseudocode for Non-Leader process.

belonging to V_2 at round 0. This can be done trivially by v_l using messages from nodes in V_1. Each node in V_1 has to simply count neighbors in V_2, the sum of these partial count is equal to $k-1$.

▶ **Theorem 13.** *Algorithm VCD solves the* ***VCDP*** *problem.*

7 InstanceCount

This algorithm assumes that the communication graph belongs to $\mathcal{G}(\mathrm{PD})_h$ then if InstanceCount notices that some process changed the distance from the leader along rounds, it invalids the count.

Non-leader process behavior (Figure 5)

Each non leader process v has three variables: $v.distance$ indicating its distance from the leader and two lists $v.M$ and $v.H$. v assigns a value to $v.distance$ as follows: if, at round r, v has $v.distance = -1$ and it is neighbor of a process with $distance = r \neq -1$, v sets its distance to $r+1$ (Line 9). Initially, the leader is the only process with $distance = 0$. As in OPT, v updates its degree history $v.H(r)$ by counting the number of processes in $N(v,r)$ whose distance is equal to $v.distance - 1$. Moreover v updates a multiset $v.M(r)$ that contains messages received by neighbors at distance $v.distance + 1$; if v has not received any of these messages, it adds $\bot$ to the multiset. In the sending phase, v broadcasts $< v.distance, v.M(r), v.H(r) >$ to its neighbors. This is done by using functions $count\_distance\_neighbors$ and $get\_messages\_from\_distance$.

A process that has $distance = r$ adds the messages from processes with $distance = -1$ to M list, let us recall that these processes with $distance = -1$ will set $distance = r+1$ at round r. Finally at Line 16 a process adds an INVCNT message to M if it detects that at least one its neighbor changed its distance from the leader which implies that the communication graph is not in $\mathcal{G}(\mathrm{PD})_h$ (see condition at Line 15). In the following when we refer to the set V_h, we consider processes setting their distance from the leader to h.

Leader process behavior (Figure 6)

The leader v_l first computes the number of processes in V_1, this is simply done by counting the messages received from these processes. After that, v_l executes VCD to count processes in V_2. This is done (i) by receiving the multi-set of messages MS from processes in V_1 (these processes are immediate neighbors of v_l) and (ii) by calling at Line 16 the function buildLastSet. This function takes the multi set MS and starts an instance of VCD to construct the multi-set MS_{last} of messages sent by processes in V_2. We define as VCD(MS, r) the local leader side simulation of a run of VCD that starts at round r using the content of messages in MS. The function returns one out of three possible values: (i) $\perp$ if the messages in MS are not enough to terminate the execution of VCD; (ii) NOCOUNT if VCD detects an halt ; (iii) A multi-set MS_{last} of messages sent by processes belonging to V_2 at round r.

This multi-set leads to the actual count of processes in V_2 (see Line 21). This procedure is iterated: each time the leader obtains the multi-set MS sent by processes in V_{h-1}, v_l calls buildLastSet to reconstruct the most recent multi-set sent by processes in V_h.

The leader returns INVCNT if either (i) there is a INVCNT message in some MS (see Lines 26) or (ii) if one of the instances of VCD terminates returning NOCOUNT. If an halt is detected then a process $v \in V_j$ at some round had a distance from v_l different than j. Additionally, at Line 8 the leader checks if processes in V_1, from which it receives messages, are stable; if this set changes the current instance is considered INVCNT.

The leader outputs the count when it counts a set V_h such that no process in V_h has a neighbor in V_{h+1}, see Line 13.

Correctness Proof

▶ **Lemma 14.** *Let R be a run of InstanceCount on a dynamic graph $G \in \mathcal{G}(PD)_h$. We have that v_l will never output INVCNT in R.*

▶ **Lemma 15.** *Let R be a run of InstanceCount on a dynamic graph $G \in \mathcal{G}(1\text{-}IC)$. If $V_h \neq \emptyset$ in R, either (1) the leader obtains the count V_h or (2) the leader outputs INVCNT.*

▶ **Lemma 16.** *Let R be a run of InstanceCount on a dynamic graph $G \in \mathcal{G}(1\text{-}IC)$. If v_l outputs a value distinct from INVCNT in R, then that value is $|V|$.*

▶ **Lemma 17.** *Let R be a run of InstanceCount on a dynamic graph $G \in \mathcal{G}(PD)_h$. We have that v_l terminate and it outputs $|V|$ in R.*

8 EXT Counting Algorithm

EXT executes an instance of InstanceCount for each temporal subgraph of G. Let us define as $\mathcal{P}_G$ as the set of such subgraphs of G. We want that processes execute for each $G' \in \mathcal{P}_G$ a different instance $I_{G'}$ of InstanceCount and that such instances do not interfere with each other. Let us remark that the system is synchronous and the current round number r is known by all processes. Therefore each $I_{G'}$ is uniquely identified by a binary string that has value 1 in position j if $G_{r_j} \in G'$ and 0 otherwise. The uniqueness guarantees that instances can run in parallel. At each new round r the number of instances is doubled, half of the new instances will consider the messages exchanged within round r and the remaining ones will not consider these messages. As example at the end round 0 we have two instances I_1, I_0. In instance I_1 the counting is started and processes have received the message exchanged in G_0. In instance I_0 the counting has not been started, the messages exchanged in round 0 are ignored. At round 1 we have four instances $I_{11}, I_{10}, I_{01}, I_{00}$: I_{11} is an instance of counting

```
$distance\_count[] = \bot$
$distance = 0$
procedure SENDING_PHASE
    send($< distance, \bot, \bot >$)

procedure RCV_PHASE($MultiSet\ MS :< distance, M, H >$)
    $i = 1$
    if $(distance\_count[i] \neq \bot \wedge distance\_count[i] \neq |MS|) \vee (\exists m \in MS | m.distance > 1)$ then
        output(INVCNT)
    $distance\_count[i] = |MS|$
    $i++$
    while $true$ do
        if $MS \neq \emptyset \wedge (\forall m \in MS : m.M = [\bot, \ldots, \bot] \wedge size(m.M) > 1)$ then
            $count = \sum_{\forall j | distance\_count[j] \neq \bot} distance\_count[j]$
            output($count$)
        $MS$ =BUILDLASTSET ($MS$)
        if $\exists$INVCNT $\in MS$ then
            output(INVCNT)
        if $MS = \bot$ then
            break
        $distance\_count[i] = |MS|$
        $i++$

function BUILDLASTSET($MS$)
    $MS_{last} = \bot$
    if MS.containsSymbol(INVCNT) then
        return {INVCNT}
    for $r$ =MinRound($MS$); $r$ <MaxRound($MS$); $r++$ do
        if VCD($MS, r$) ==NOCOUNT then
            return {INVCNT}
        if VCD($MS, r$) $\neq \bot$ then
            if $MS_{last} \neq \bot \wedge |MS_{last}| \neq |$ VCD($MS, r$)$|$ then
                return {INVCNT}
            $MS_{last}$ = VCD($MS, r$)
        else
            break
    return $MS_{last}$
```

Figure 6 InstanceCount for $\mathcal{G}(1\text{–IC})$: pseudocode for Leader process.

in which messages exchanged in G_0, G_1 are considered; in I_{10} are considered only messages exchanged in G_1 and ignored messages exchanged in G_0; in I_{01} are considered only messages exchanged in G_0 and ignored messages exchanged in G_1; in I_{00} the counting has not been started. The pseudocode to implement the this procedure is trivial, thus it is omitted.

▶ **Theorem 18.** *Let R be a run of EXT on a dynamic graph $G \in \mathcal{G}$(1-IC). Eventually, v_l terminates and it outputs the correct count in R.*

From the previous Theorem and from the impossibility of non trivial computation without a leader presented in [18, 19] we have:

▶ **Theorem 19.** *Let us consider an anonymous unknown 1-interval connected networks with broadcast. A distinguished leader process is necessary and sufficient to do non trivial computations.*

Besides counting and existence predicates other non-trivial problems are solvable using simple variation of EXT. Let us assume that each process has an initial input value. If this initial input is attached in the messages of EXT the leader can compute the exact multiset of these values. Thanks to this multiset the leader may compute aggregation functions as average,min,max.

Complexity Discussion

The EXT algorithm has an exponential complexity:. If we consider that distances of each node from v_l are in $[1, |V|-1]$, then it is easy to see that the number of possible combinations of distances over the set of nodes is upper bounded by $|V|^{|V|}$, therefore by definition of $\mathcal{G}(\text{PD})$ we have $max_j(|i_j - i_{j+1}|) \leq |V|^{|V|}$. Now what we have to bound is the number of instances of G' needed by EXT to terminate, but this can be easily computed by considering when counting terminate with $\mathsf{InstanceCount}$ on a graph $\mathcal{G}(\text{PD})$. At each level we count in at most $\mathcal{O}(|V|^3)$ rounds, therefore it is easy to show by straightforward induction that the total cost is $\mathcal{O}(|V|^4)$. So EXT terminates in at most $\mathcal{O}(|V|^{|V|+4})$ rounds.

9 Conclusion

In this paper we have shown that, in anonymous interval connected network with broadcast, a leader node is enough to do non trivial computations. This answers negatively to the conjecture presented in [19, 18]. Moreover we have shown an optimal counting algorithm for $\mathcal{G}(\text{PD})_2$ networks, proving the tightness of the bound shown in [14]. However, our EXT algorithm has an exponential complexity, both in memory and in the number of rounds. In $\mathcal{G}(\text{1-IC})$ networks with IDs, when there is unlimited bandwidth, counting requires $O(|V|)$ rounds. It is unknown if handling anonymity in $\mathcal{G}(\text{1-IC})$ requires this exponential cost. A future line of work could be the investigation of this gap.

References

1 D. Angluin. Local and global properties in networks of processors (extended abstract). In *STOC'80*, pages 82–93. ACM, 1980. doi:10.1145/800141.804655.

2 R. Baldoni, S. Bonomi, A. Kermarrec, and M. Raynal. Implementing a register in a dynamic distributed system. In *IEEE International Conference on Distributed Computing Systems (ICDCS'09)*, pages 639–647, 2009. doi:10.1109/ICDCS.2009.46.

3 M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. *J. Comput. Syst. Sci.*, 73(3):245–264, May 2007. URL: 10.1016/j.jcss.2006.10.007, doi:10.1016/j.jcss.2006.10.007.

4 Joffroy Beauquier, Janna Burman, Simon Clavière, and Devan Sohier. Space-optimal counting in population protocols. In *(to appear) DISC'15*, 2015. URL: https://hal.inria.fr/hal-01169634.

5 P. Boldi and S. Vigna. Computing anonymously with arbitrary knowledge. In *PODC'99*, pages 181–188. ACM, 1999.

6 P. Boldi and S. Vigna. Fibrations of graphs. *Discrete Mathematics*, 243(1-3):21–66, 2002. doi:10.1016/S0012-365X(00)00455-6.

7 A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *CoRR*, abs/1012.0009, 2010. URL: http://arxiv.org/abs/1012.0009.

8 P. Fraigniaud, A. Pelc, D. Peleg, and S. Pérennes. Assigning labels in an unknown anonymous network with a leader. *Distributed Computing*, 14(3):163–183, 2001. doi:10.1007/PL00008935.

9 M. Jelasity, A. Montresor, and Ö. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

10 D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *FOCS'03*, pages 482–491. IEEE, 2003. doi:10.1109/SFCS.2003.1238221.

11 J. Kong, X. Hong, and M. Gerla. An identity-free and on-demand routing scheme against anonymity threats in mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 6(8):888–902, 2007.

12 F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *STOC'10*, pages 513–522. ACM, 2010. doi:10.1145/1806689.1806760.

13 F. Kuhn and R. Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, March 2011. doi:10.1145/1959045.1959064.

14 G. Di Luna and R. Baldoni. Brief announcement: Investigating the cost of anonymity on dynamic networks. In *PODC'15*, pages 339–341. ACM, 2015. doi:10.1145/2767386.2767442.

15 G. Di Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. Conscious and unconscious counting on anonymous dynamic networks. In *ICDCN'14*, pages 257–271. Springer, 2014.

16 G. Di Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. Counting in anonymous dynamic networks under worst case adversary. In *ICDCS'14*, pages 338–347. IEEE, 2014.

17 L. Massoulié, E. Le Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: Random walk methods. In *PODC'06*, pages 123–132. ACM, 2006. doi:10.1145/1146381.1146402.

18 O. Michail, I. Chatzigiannakis, and P. Spirakis. Brief announcement: Naming and counting in anonymous unknown dynamic networks. In *DISC'12*, pages 437–438. Springer, 2012.

19 O. Michail, I. Chatzigiannakis, and P. Spirakis. Naming and counting in anonymous unknown dynamic networks. In *SSS'13*, pages 281–295. Springer, 2013. doi:10.1007/978-3-319-03089-0_20.

20 O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Causality, influence, and computation in possibly disconnected synchronous dynamic networks. In *OPODIS'12*, pages 269–283, 2012. doi:10.1007/978-3-642-35476-2_19.

21 D. Mosk-Aoyama and D. Shah. Computing separable functions via gossip. In *PODC' 06*, pages 113–122. ACM, 2006. doi:10.1145/1146381.1146401.

22 R. O'Dell and R. Wattenhofer. Information dissemination in highly dynamic graphs. In *DIALM-POMC' 05*, pages 104–110, 2005. doi:10.1145/1080810.1080828.

23 B. Ribeiro and D. Towsley. Estimating and sampling graphs with multidimensional random walks. In *IMC'10*, pages 390–403, New York, NY, USA, 2010. ACM. doi:10.1145/1879141.1879192.

24 N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC'99, pages 173–179. ACM, 1999. doi:10.1145/301308.301352.

25 M. Yamashita and T. Kameda. Computing on an anonymous network. In *PODC'88*, pages 117–130. ACM, 1988. doi:10.1145/62546.62568.

26 M. Yamashita and T. Kameda. Computing on anonymous networks: Part 1-characterizing the solvable cases. *IEEE Trans. on Parallel and Distributed Systems*, 7(1):69–89, 1996. doi:10.1109/71.481599.

Analysis of Bounds on Hybrid Vector Clocks*

Sorrachai Yingchareonthawornchai[1], **Sandeep Kulkarni**[2], **and Murat Demirbas**[3]

1 **Michigan State University, MI, USA**
`yingchar@cse.msu.edu`

2 **Michigan State University, MI, USA**
`sandeep@cse.msu.edu`

3 **University at Buffalo, SUNY, NY, USA**
`demirbas@buffalo.edu`

Abstract

Hybrid vector clocks (HVC) implement vector clocks (VC) in a space-efficient manner by exploiting the availability of loosely-synchronized physical clocks at each node. In this paper, we develop a model for determining the bounds on the size of HVC. Our model uses four parameters, ϵ: uncertainty window, δ: minimum message delay, α: communication frequency and n: number of nodes in the system. We derive the size of HVC in terms of a differential equation, and show that the size predicted by our model is almost identical to the results obtained by simulation. We also identify closed form solutions that provide tight lower and upper bounds for useful special cases.

Our model and simulations show the HVC size is a sigmoid function with respect to increasing ϵ; it has a slow start but it grows exponentially after a phase transition. We present equations to identify the phase transition point and show that for many practical applications and deployment environments, the size of HVC remains only as a couple entries and substantially less than n. We also find that, in a model with random unicast message transmissions, increasing n actually helps for reducing HVC size.

1998 ACM Subject Classification C.2.4 Distributed Systems, distributed databases, D.1.3 Concurrent Programming, D.4.2. Distributed memories, D.4.3 Distributed file systems

Keywords and phrases Vector Clocks, Physical Clocks, Large Scale Systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.34

1 Introduction

Work on theory of distributed systems abstract away from the wall-clock/physical-clock time and use the notion of logical clocks for ordering events in *asynchronous* distributed systems [12, 10, 13]. The causality relationship captured by these logical clocks, called happened-before (**hb**), is defined based on passing of information, rather than passing of time.[1] Lamport's logical clocks [12] (LC) prescribe a total order on the events: $A \ \underline{hb}\ B \implies lc.A < lc.B$ but vice a versa is not necessarily true. Vector clocks [10, 13] (VC) prescribe a partial order on the events: $A \ \underline{hb}\ B \iff vc.A < vc.B$ and $A \ \underline{co}\ B \iff (\neg(vc.A < vc.B) \wedge \neg(vc.B < vc.A)$. Using LC or VC, it is not possible to query events in relation to physical time. Moreover, for capturing **hb**, LC and VC assume that all communication occur

* This work is supported by NSF CNS 1329807, NSF CNS 1318678, NSF XPS 1533870, and XPS 1533802.

[1] Event A **hb** event B, if A and B are on the same node and A comes earlier than B, or A is a send event and B is the corresponding receive event, or this is defined transitively based on the previous two.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 34; pp. 34:1–34:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in the present system and there are no backchannels. This assumption is obsolete for today's integrated, loosely-coupled system of systems. Finally, the space requirement of VC is shown to be $\Theta(n)$ [3], the number of nodes in the system, and is prohibitive.

Practice of distributed systems, on the other hand, employ loosely synchronized clocks, mostly using NTP [15]. Unfortunately, there are fundamental limits to clock synchronization and perfect synchronization is unachievable due to the nature of distributed systems: messaging with uncertain latency, clock skew among processors, and NTP glitches [15]. Even using atomic clocks, as in Google TrueTime [5], it is hard to reduce ϵ, the uncertainty of the clock synchronization, to less than a couple milliseconds. This requires that operations/transactions wait out these ϵ uncertainties, which takes its toll on the performance.

Recently, we introduced a third option, hybrid clocks [6, 11]. Hybrid clocks combine the best of logical and physical clocks; hybrid clocks are immune to their disadvantages while providing their benefits. Hybrid clocks are loosely synchronized using NTP, yet they also provide provable comparison conditions as in LC or VC within ϵ uncertainty. Hybrid clocks also address the backchannel communication issue by introducing the notion of $\epsilon - \underline{hb}$ that captures the intuition that if event B happened far later than event A, then event A can affect event B due to out-of-bound communication. If events A and B are close, then the causality relation is taken into account to identify whether A can affect B.

Our hybrid clocks come in two flavors: hybrid logical clocks (HLC) [11] and hybrid vector clocks (HVC) [6]. HLC satisfy the logical clock comparison condition as in LC [12]. HLC finds applications in multiversion distributed database systems [4] and enable efficient querying of consistent snapshots for read transactions, while ensuring commits of write transactions do not get delayed despite the uncertainties in NTP clock synchronization [11]. HVC satisfy the vector clock comparison condition as in VC [10, 13], and can serve in applications that HLC become inadequate. In contrast to HLC that can provide a single consistent snapshot for a given time, HVC is able to provide all possible/potential consistent snapshots for that given time. As such, HVC finds applications in debugging for concurrency race conditions of safety critical distributed systems and in causal delivery of messages to distributed system nodes.

HVC reduces the overhead of causality tracking in VC by utilizing the fact that the clocks are reasonably synchronized. When ϵ is infinity, HVC behaves more like VC used for causality tracking in asynchronous distributed systems. When ϵ is very small, HVC behaves more like a scalar physical synchronized clock, but also combines the benefits of causality tracking in uncertainty intervals. Although the worst case size for HVC is $\Theta(n)$, we observe that if j does not hear (directly or transitively) from k within ϵ time then $hvc.j[k]$ need not be explicitly maintained. In that case, we still infer implicitly that $hvc.j[k]$ equals $hvc.j[j] - \epsilon$, because $hvc.j[k]$ can never be less than $hvc.j[j] - \epsilon$ thanks to the clock synchronization assumption. Therefore, in practice the size of $hvc.j$ would only depend on the number of nodes that communicated with j within the last ϵ time and provided a fresh timestamp that is higher than $hvc.j[j] - \epsilon$. In other words, by using temporal slicing, HVC can circumvent the Charron-Bost result [3] and can potentially scale the VC benefits to many thousands of processes by still maintaining small HVC at each process.

Contributions of this paper. But how effective are HVC for reducing the size of VC? What bounds should we expect on the number of entries in HVC for a given ϵ? Determining these bounds on HVC would help developers to budget the size of the messages the nodes send, the size of the memory to maintain at the nodes, and the scalability and performance of their system. In this paper, we derive and identify these bounds.

To this end, we develop an analytical model that uses four parameters, ϵ: uncertainty window, δ: minimum message delay, α: message rate, and n: number of nodes in the system.

We derive the size of HVC in terms of a differential equation, and show that the size predicted is almost identical to the results obtained by simulation experiments. We also identify closed form solutions that provide tight lower and upper bounds for useful special cases.

Our model and simulations show the HVC size is a sigmoid function with respect to increasing ϵ; it has a slow start but it grows exponentially after a critical phase transition. Before the phase transition threshold, HVC maintains couple entries per node, however when a threshold is crossed, a node not only gets entries added to its clock from direct interaction but also indirect transfer from another processes HVC, and this makes the HVC entries blow up. We present equations to identify this transition point. Specifically, for the common case of $\alpha * \delta < 1$, we derive this threshold as $(\frac{1}{\alpha} + \delta)(\ln((2 - \sqrt{3})(n - 1)))$.

Using this equation, we describe how to avoid/delay the threshold point. If an application developer reduces α, the phase transition is delayed, and small HVC sizes are still achievable for a given ϵ and δ. Moreover, while in VC the size increases directly with n, we find that in HVC, surprisingly, the increase of n, in fact, benefits in reducing the size of HVC. Using a model with random unicast message transmissions, for larger n, the probability of indirect HVC entry addition/transfer reduces slightly, and hence larger n, in fact delays the phase transition to large HVC sizes.

We show in our discussion section that for most practical applications and deployment environments, the size of HVC remains only as a couple entries and substantially less than n. Yet, when it is needed HVC expands on demand to allow more entries to capture causality both ways in the ϵ uncertainty slices.

Outline of the rest of the paper. After presenting the preliminaries in Section 2, we present our analytical solutions in Section 3, and solutions for useful special cases in Section 4. We present evaluation results in simulation to show how well the analytical models capture the HVC bounds in Section 6. We discuss practical implications of our findings in Section 7, related work in Section 8, and conclude in Section 9.

2 System Model

We use n to denote the number of processes in the system. Although processes can be added dynamically, we assume that each of them has a distinct identifier. Each process j is associated with a physical clock $pt.j$. We assume that clock synchronization algorithm such as NTP [15] is used to provide a reasonable but imperfect clock synchronization to the processes. For ease of presentation, we assume the existence of an *absolute time*: this time is not accessible to processes themselves, and it is used only for the presentation and proofs associated with our algorithm. Specifically, we assume that at any given time the difference between any two clocks at processes, $pt.j$ and $pt.k$, is bounded by $\epsilon, \epsilon \geq 0$.

Processes communicate via messages. We make no assumptions such as FIFO ordering or bounded delivery time. In other words, messages could be delivered out of order. They could also be delivered a long time after they are sent. We assume that there is a *minimum message delay* δ_{min} (as computed by the absolute global time) before message is delivered.

In our analytical model to compute the size of HVC at any process, we assume that at each absolute time tick, each process sends a message to some other process (selected randomly) with probability α. We permit messages to be delivered as early as possible, we allow a process to receive multiple messages simultaneously.

Let $s_j(t)$ be a random variable representing size of active HVC of process j at time t. Thus, our goal is to identify an expected average active HVC size $\Psi(t) = E[\sum_j s_j(t)/n]$, and

$\psi(t) = \Psi(t)/n$. We aim to find an analytical solution to $\psi(t)$ given four parameters ϵ, δ, α, and n.

2.1 Unconstrained and Constrained Time Models

To develop this analytical solution, we develop two models: 1) an unconstrained model where we compute the size of HVC by assuming that $\epsilon = \infty$, and 2) a constrained model that considers the value of ϵ. Without loss of generality, we focus on one sender process, say j. Our goal is to identify the number of processes that maintain he clock of this process at a given time t. In turn, this enables us to find the expected size of each HVC entry. To make this analysis simpler to understand, we introduce the notion of a color –red or green– for each process. The color of process k is red at time t iff k is maintaining the clock of process j at time t. In other words, *color.k* is red iff the knowledge that k has about the clock of j is more than that provided by clock synchronization. Clearly, in the initial state $t = 0$, j is red and all other processes are green.

Model 1: unconstrained time model. Given the notion of color maintained by each process, we can observe that if a red process sends a message to a green process, then the green process learns information about the clock of j. In other words, it makes the recipient red. Messages sent by green process can be ignored since they do not provide non-trivial information about the clock of j.

In this model, let $Y(t)$ denote number of red processes at time t. Note that $n - Y(t)$ is number of green processes at time t. Also, let $y(t) = Y(t)/n$ be the fraction of red process at time t. We aim to analytically compute $y(t)$ given δ, α, n for $\epsilon \to \infty$.

Model 1 captures the case where $\epsilon = \infty$. The reason we consider this model is due to an important result (shown in Theorem 13) that demonstrates that the value of $Y(\epsilon)$ can be used to compute the number of red processes in the ϵ-constrained model (discussed next) that utilizes the actual value of ϵ in the given system.

Model 2: ϵ-constrained time model. To capture the effect of the hybrid model where ϵ has a finite value (and hence, a red process will turn green if it does not hear recent clock information of process j), we define τ-message as a message that is originated by the initial red process j at time τ. τ-message triggers green process to be red if $\tau + \epsilon \leq t$. Otherwise, even if the green process receives information about the clock of j, this information is still beyond the uncertainty interval. Let $Y_\epsilon(t)$ be number of red processes of Model 2 at time t. We aim to compute an analytical solution to $y_\epsilon(t) = Y_\epsilon(t)/n$ for given ϵ, δ, α, and n.

3 Analytical Solutions

Given that ϵ-constrained time model can be answered by unconstrained time model as shown in Theorem 13, this means analytical solution to unconstrained time model implies the solution to our system model.

Based on the definition of *color.k*, in the initial state, *color.j* is red and *color.k* is green for any $k \neq j$. It follows that at time $t = [0..\delta]$, j is the only red process as message sent by j has not been received by anyone. When a green process receives a message it turns red and stays red forever. Let $Y(t)$ be number of red processes at time t. Note that number of green processes at time t is then $n - Y(t)$. Since message delay for every message is δ, $Y(t)$ depends upon $Y(t - \delta)$, i.e., the number of processes that were red δ time before.

Our first result in this context, given in Lemma 1, captures the number of messages delivered at time t to green processes.

▶ **Lemma 1.** *The expected number of messages delivered to green processes at time t is $\alpha Y(t-\delta)(1-Y(t)/n)$*

Proof. The expected number of red messages delivered at time t is $\alpha Y(t-\delta)$ since each red process in $Y(t-\delta)$ has α probability to send a unicast message. At time t, the probability of a message getting delivered to green is the fraction of green process at time t, $1-Y(t)/n$ assuming that each process has equally likely change to receive such message. The result follows immediately by linearity of expectation. ◀

Although Lemma 1, counts the number of red messages sent to green processes, it overcounts the processes that can become red, as one green process may receive multiple messages. To analyze the number of processes that turn red, we observe that this problem can be viewed as throwing a number of balls (i.e., messages sent by red processes) into a set of bins (i.e., the green processes) to identify the expected number of non-empty bins (i.e., processes that receive at least one ball and therefore turn red). In this context, we use Lemma 2.

▶ **Lemma 2.** *Consider occupancy problem where there are A balls and B bins. All balls are thrown to random bins. Expected number of non-empty bins is $B(1-(1-1/B)^A)$.*

Proof. Fix one bin. Probability of the bin being empty is $(1-1/B)^A$ since all balls must miss this bin. By linearity of expectation, expected number of empty bins is $B(1-1/B)^A$. Hence, expected number of non-empty bins is B minus number of empty bins. ◀

Now, we can compute the change of red processes at time t by applying Lemma 2 using $A = \alpha Y(t-\delta)(1-Y(t)/n)$ (from Lemma 1) and $B = n - Y(t)$ since B is number of green process at time t. Hence, $\frac{dY(t)}{dt}$ is $(n-Y(t))(1-(1-\frac{1}{(n-Y(t))})^{\alpha Y(t-\delta)(1-Y(t)/n)})$

We can simplify the expression by using the fact that $\lim_{n\to\infty}(1+x/n)^n = e^x$. We adjust some terms in Equation above and let $x = \frac{-1}{(1-Y(t)/n)}$, we obtain

$(n-Y(t))(1-(1-\frac{1}{n(1-Y(t)/n)})^{\alpha\frac{n}{n}Y(t-\delta)(1-Y(t)/n)}) = (n-Y(t))(1-e^{\frac{-\alpha Y(t-\delta)}{n}})$

Since $y(t) = Y(t)/n$, we get $\frac{dy(t)}{dt} = (1-y(t))(1-e^{-\alpha y(t-\delta)})$

Finally, based on the initial values, we have $y(t) = 1/n$ for $t < \delta$. And, since we can consider each process j independently, which means the expectation does not change. Thus, we have the following Theorem.

▶ **Theorem 3.** *The expected average size of hvc per process of $\psi(t)$ satisfies the following delay differential equation.*

$$\frac{d\psi}{dt} = (1-\psi(t))(1-e^{-\alpha\psi(t-\delta)})$$

where initial condition is $\psi(t) = 1/n$ for $t < \delta$.

From this point on, we use $\psi(t)$ (random variable of fraction of average size of *hvc*) and $y(t)$ (random variable of fraction of red processes) interchangably since they have same expactation value.

4 Explicit Solutions for Special Cases

Theorem 3 provides a mechanism to compute the size of *hvc*. Since the differential equation in Theorem 3 cannot be solved explicitly, one must utilize numerical tools, such as MATLAB and Mathematica, to obtain the size of *hvc* from that equation.

However, closed form solutions —that can be computed with a basic calculator— may be more desirable since they can offer a quick insight into the size of *hvc*.

In this section, we provide closed form solutions for some special cases. Specifically, when α is arbitrarily small, we obtain an explicit solution to Theorem 3 given that $\alpha * \delta$ is small. If $\alpha * \delta$ is not necessarily small, we derive an yet explicit solution up to $\epsilon \leq 3\delta$ for arbitrary δ. Using simplification technique, we can obtain upper and lower bounds solution to Theorem 3 if α is not necessarily small. Based on our evaluation, the value of $\alpha * \delta < 1$ is sufficient to obtain accurate closed form solutions. Otherwise, $\epsilon \leq 3\delta$ can capture almost all value of y.

These bounds are fairly tight as shown in the simulation results in Section 6. The problem for computing closed form solution where $\delta > 0$ and $\epsilon > 3\delta$ is currently open.

4.1 Explicit Solution for Arbitrarily Small α and $\alpha * \delta$

We put two main simplifications to obtain explicit solutions. First, we assume that α is small (typically, $\alpha < 0.1$) so that we have good approximation of $1 - e^{-\alpha y(t-\delta)}$ using Taylor's series expansion. The expansion is $\alpha y - \alpha^2 y^2/2 + O(\alpha^3 y^3)$. If α is small, this expansion is approximately αy. Hence, the differential equation in Theorem 3 becomes $\frac{dy}{dt} = \alpha(1 - y(t))y(t - \delta)$. Second simplification is suppose $\alpha * \delta$ that is arbitrarily small. We have the following Lemma.

▶ **Lemma 4.** *if $\alpha\delta > 0$ is arbitrarily small, then $y(t) = (1 + \alpha\delta)y(t - \delta)$*

Proof. We can approximate the change of $y(t)$ over δ period of time in the past. That is, the change $\frac{y(t)-y(t-\delta)}{\delta}$ is approximately $\frac{dy}{dt}$. Based on expression above, the change is roughly $\alpha(1 - y(t))y(t - \delta)$. Therefore, $y(t) = y(t - \delta) + \alpha\delta(1 - y(t))y(t - \delta)$. The result follows from that the product $\alpha\delta$ is approaching zero. ◀

Using Lemma above, we reduce delay differential equation to ordinary differential equation as in the following. The differential equation is elementary to be solved by standard ordinary differential equation procedure.

▶ **Theorem 5.** *For the case where $\alpha * \delta > 0$ is arbitrarily small, the change of y over time is $\frac{dy}{dt} = \frac{\alpha}{1+\alpha\delta}(1 - y)y$ with initial condition $y(0) = 1/n$. Further, the explicit solution to the differential equation is*

$$y(t) = \frac{1}{1 + (n - 1)e^{-\alpha t/(1+\alpha\delta)}} .$$

4.2 Phase Transition

The result for Theorem 5 implies that the graph of y is essentially a logistic function (or Sigmoid function). One important characteristic of this function is it has slow start in the initial state and then the function grows exponentially after a phase transition. In this section, we discover such transition in terms of δ, α and n. We define phase transition point ϵ_p as the earliest point where the change of slope is maximum. In particular, we show the following result.

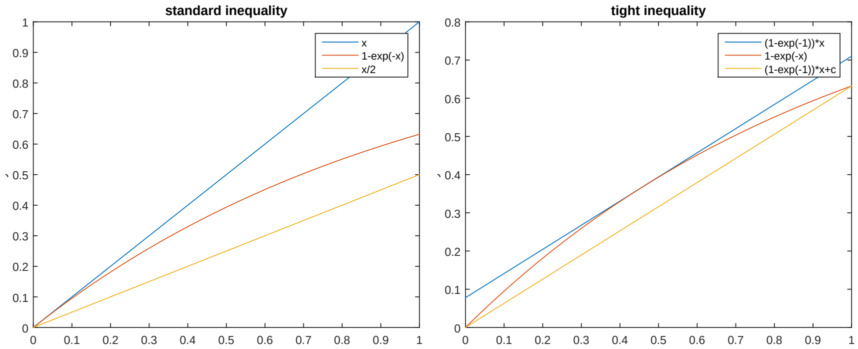


Figure 1 Standard inequality and tight inequality.

▶ **Theorem 6.** *The phase transition* ϵ_p *for Theorem 5 is* $(\frac{1}{\alpha}+\delta)(\ln((2-\sqrt{3})(n-1)))$.

Proof. The slope of $y(t)$ is $y'(t)$. The change of slope is $y''(t)$. The maximum of change of slope is when $y^{(3)}(t) = 0$. We get the result by finding third order derivative of y. Then, we set y''' to 0. Suppose the function is in the following form: $\frac{dy}{dt} = a(1-y)y$. We apply derivative twice from $\frac{dy}{dt}$ to obtain the third order derivative of y. By simple differentiation, we have

$$y'''(t) = \frac{a^3(n-1)e^{at}(-4(n-1)e^{at} + e^{2at} + (n-1)^2)}{(e^{at}+n-1)^4}\,.$$

When $y'''(t) = 0$, we obtain quadratic equation in the form of e^{at}:

$$e^{2at} - 4(n-1)e^{at} + (n-1)^2 = 0\,.$$

Solving quadratic quation, we obtain $e^{at} = (n-1)(2 \pm \sqrt{3})$. We select the earlier time by definition of phase transition. Then, $t = (1/a)(\ln((n-1)(2-\sqrt{3}))$. The result follows when we substitute $a = \frac{\alpha}{1+\alpha\delta}$. ◀

4.3 Explicit Solution for $t < 3\delta$

If α is not necessarily small, we can obtain bounds in terms of upper and lower bounds. The technique is to simplify the function so that differential equation is easily solvable. Since the equation in Theorem 3 involves e^x, in Lemma 7, we first identify a tight bound on the value of e^x when x is in the range $[0..1]$.[2]

▶ **Lemma 7.** *For* $x, \alpha \in [0,1]$, *this inequality holds*

$$(1-e^{-\alpha})x \le 1 - e^{-\alpha x} \le (1-e^{-\alpha})x + \xi$$

where

$$\xi = 1 - (\frac{1-e^{-\alpha}}{\alpha})(1 + \ln(\frac{\alpha}{1-e^{-\alpha}}))\,.$$

[2] **Remark.** The standard inequality identity regarding e^x is that $1 - e^{-x} \le x$ for any real number x, and $x/2 \le 1 - e^{-x}$ for some small range x. We considered using these upper and lower bounds in subsequent results. However, these bounds are not tight when $x \in [0..1]$ as shown in Figure 1, which is the case in Theorem 3. This is the reason we use Lemma 7 in subsequent computation.

Proof. We only need to find slope and y-intercept for two lines. The lower bound is easily attainable by considering two points $(0, 0)$ and $(1, 1 - e^{-\alpha})$. For the upper line, we know that the slope must be equal to the lower line, which is $1 - e^{-\alpha}$. We want the upper line to touch exactly one point above the function $1 - e^{-\alpha x}$ in some point $x \in [0, 1]$. The only remaining part is to find y-intercept. First, we find a point of the function $1 - e^{-\alpha x}$ such that the line passing it has slope of $1 - e^{-\alpha x}$. Using basic derivative and solve for x we get $x = \frac{1}{\alpha}\ln(\frac{\alpha}{1-e^{-\alpha}})$

Substituting x in the function $1 - e^{-\alpha x}$ yields $1 - (\frac{1-e^{-\alpha}}{\alpha})$. Finally, we find y-intercept of upper line given slope of $1 - e^{-\alpha}$.

$$\begin{aligned} y &= mx + c \\ 1 - (\frac{1 - e^{-\alpha}}{\alpha}) &= \frac{(1 - e^{-\alpha})}{\alpha}(\ln(\frac{\alpha}{1 - e^{-\alpha}})) + c \\ c &= 1 - (\frac{1 - e^{-\alpha}}{\alpha})(1 + \ln(\frac{\alpha}{1 - e^{-\alpha}})) \end{aligned}$$ ◀

Subsequently, we use the upper and lower bounds identified in Lemma 7 in Theorem 3 for the case where δ is arbitrary but $t \le 3\delta$. In other words, this allows us to capture how the size of *hvc* grows in the first 3δ time. This gives us another explicit function if $\alpha * \delta > 1$ and is evaluated in the Simulation section. Note that the bound in Lemma 7 is quite tight as we can see the result presented in Section 6.3.

▶ **Theorem 8.** *The solution $\psi(t)$ to Theorem 3 is bounded by the following time condition.*

- *For $t \in [\delta, 2\delta]$,*

  $$\psi(t) = 1 - ke^{-\alpha t/n}$$

  where $k = (1 - 1/n)e^{\alpha\delta/n}$.
- *For $t \in [2\delta, 3\delta]$,*

  $$1 - k_\ell H(t) \le \psi(t) \le 1 - k_u H(t)e^{\xi(t-\delta)}$$

  where $H(t) = e^{(1-e^{-\alpha})(kne^{-\alpha(t-\delta)/n}/\alpha+t-\delta)}$ and

  $$\begin{aligned} k_\ell &= (1 - (1 - ke^{-2\delta\alpha/n}))e^{(1-e^{-\alpha})(\frac{kn}{\alpha}e^{-\delta\alpha/n}+\delta)} \\ k_u &= k_\ell e^{\delta\xi} \\ \xi &= 1 - (\frac{1 - e^{-\alpha}}{\alpha})(1 + \ln(\frac{\alpha}{1 - e^{-\alpha}})) \end{aligned}$$

Proof. For $t \in [\delta, 2\delta]$, we can model as a sequence of single unicast message from the past $t - \delta$ and quantify the change accordingly. During $t \in [\delta, 2\delta]$, there is at most one message delivered because during $t - \delta$ there is only one green process, i.e., process j. Therefore, at any time the change of y depends only current y and one message with probability α. The expected change of fraction of Y over time is given a simple differential equation: $\frac{dy}{dt} = \frac{\alpha}{n}(1 - y)$ with initial condition $y(\delta) = 1/n$. Solving ordinary differential equation is an easy exercise.

▶ **Lemma 9.** *The solution to differential equation: $\frac{dy}{dt} = \frac{\alpha}{n}(1 - y)$ with initial condition $y(\delta) = 1/n$ is $y_1(t) = 1 - ke^{-\alpha t/n}$ where $k = (1 - 1/n)e^{\alpha\delta/n}$.*

For $t \in [2\delta, 3\delta]$, we replace the term $1 - e^{-\alpha y}$ with corresponding lower and upper bounds in Lemma 7. Consider the delay differential equation in Theorem 3 $\frac{dy}{dt} = (1-y)(1-e^{-\alpha y(t-\delta)})$

During $t \in [2\delta, 3\delta]$, the function $y(t-\delta)$ becomes $y_1(t-\delta) = 1 - ke^{-\alpha t/n}$ By Lemma 9. We use the tight lower and upper bounds as in Lemma 7 to obtain the equation:

$$(1-y)Ay \leq \frac{dy}{dt} \leq (1-y)(Ay+\xi)$$

where $A = (1-e^{-\alpha})$. Then, we instantiate value of $y = y_1 = 1 - ke^{-\alpha(t-\delta)/n}$ to obtain the following inequality:

$$(1-y)((1-e^{-\alpha})(1-ke^{-\alpha(t-\delta)/n})) \leq \frac{dy}{dt} \leq (1-y)((1-e^{-\alpha})(1-ke^{-\alpha(t-\delta)/n}) + \xi)\,.$$

From this expression, we can consider only the equation $\frac{dy}{dt} = (1-y)((1-e^{-\alpha})(1-ke^{-\alpha(t-\delta)/n}) + \xi)$ as the lower bound term follows immediately from upper bound result when instantiating $b = 0$ of Lemma 10, which is easily verified.

▶ **Lemma 10.** *We have the following integral results*

$$\int ((1-e^{-\alpha})(1-ke^{-\alpha(t-\delta)/n}) + b)dt = (1-e^{-\alpha})(\frac{kn}{\alpha}e^{-\alpha(t-\delta)/n} + (t-\delta)) + b(t-\delta) + C\,.$$

The remaining part is to use the result from Lemma 10 to solve ordinary differential equation and find a constant term with initial condition $y(2\delta) = 1 - ke^{-2\delta\alpha/n}$.

$$\begin{aligned}
\frac{dy}{dt} &= (1-y)((1-e^{-\alpha})(1-ke^{-\alpha(t-\delta)/n}) + \xi) \\
\int (\frac{1}{1-y})dy &= \int ((1-e^{-\alpha})(1-ke^{-\alpha(t-\delta)/n}) + b)dt \\
y &= 1 - k_0 e^{-((1-e^{-\alpha})(\frac{kn}{\alpha}e^{-\alpha(t-\delta)/n} + t - \delta) + b(t-\delta))}
\end{aligned}$$

The results follow since k_0 can be solved with initial condition $y(2\delta) = 1 - ke^{-2\delta\alpha/n}$. This completes the proof. ◀

5 Reduction of ϵ-Constrained Time Model to Unconstrained Time Model

In this section, we show that unconstrained and ϵ-constrained time model are closely related. In particular, in Theorem 13, we show that ϵ-constrained time model can be solved by the solution for the unconstrained time model.

We first describe the basic idea behind Theorem 13. Initially, the process j is the only one red process. After some time, the number of red processes increases since process j sends message to some other processes and other processes that carry active information about j also send this entry, i.e., red processes help disseminate red messages. At the same time, if a process does not hear a message that contains newer information (directly or indirectly) about process j then in ϵ-constrained time model, this process should turn green. Therefore, at any time, the change of number of red processes is due to (1) green processes turning red, and (2) red processes turning green. We show that the number of red processes remains unchanged after some period of time. That is, the increase due to (1) is equal to the decrease due to (2), i.e., it reaches an equilibrium point.

To prove our result about ϵ-constrained time model (i.e., Model 2 in Section 2), we put different time labels on color. A process is τ-red if it receives τ-message directly or transitively, i.e., a message that is originated from process j at time τ. A process is red at time t if and only if it is τ-red for some $\tau \in [t-\epsilon, t]$.

Let $r_\tau(t)$ be a set of τ-red processes at time t. Based on definition of $r_\tau(t)$, we can compute the cardinality of $r_\tau(t)$.

▶ **Lemma 11.** *The expected number of τ-red processes is given by*

$$E[|r_\tau(t)|] = \begin{cases} 0 & \text{if } t \leq \tau \text{ or } t > \tau + \epsilon \\ y(t-\tau) & \text{otherwise} \end{cases}$$

Proof. If $t \leq \tau$ or $t > \tau + \epsilon$, it is either τ-message non-existent or expired. Otherwise, at time τ, process j sends first τ-message. This time is the initial condition of $y(t)$ which is $y(0)$. Thereafter, the number of τ-red process is equivalent to that of unconstrained time model since the τ-message is not expired until $t > \tau + \epsilon$. Hence, the result follows. ◀

▶ **Corollary 12.** *The following equation holds $|r_\tau(t)| = |r_{\tau+1}(t+1)|$ with high probability.*

Proof. The expression $E[|r_\tau(t)|] = E[|r_{\tau+1}(t+1)|]$ holds by simply substituting t as $t+1$ and τ as $\tau+1$ in the Lemma 11. ◀

Using these two results, we show that the fraction of the red processes in the ϵ-constrained time model can be derived by using the unconstrained time model as follows:

▶ **Theorem 13.** *Let $y(t), y_\epsilon(t)$ be fraction of red process at time t from model 1 and 2 respectively. $y_\epsilon(t)$ can be computed by the following expression.*

$$y_\epsilon(t) = \begin{cases} y(t) & \text{if } t \leq \epsilon \\ y(\epsilon) & \text{otherwise} \end{cases}$$

Proof. Define $R(t)$ as a set of red processes at time t. This is basically a union of τ-red processes at time t for $t - \epsilon \leq \tau \leq t-1$. That is, $R(t) = \bigcup_{i=t-\epsilon}^{t-1} r_i(t)$. We note that $r_{\leq 0}(t) = \emptyset$ by definition. Hence, $R(i)$ for $0 \leq i \leq \epsilon - 1$ collects more term until $i \geq \epsilon$, which follows terms from definition of $R(t)$.

We show that expectation of $E[|R(t)|] = E[|R(t+1)|]$ for $t \geq \epsilon$. By definition, observe that $R(t+1) = \bigcup_{i=t-\epsilon+1}^{t} r_i(t+1) = \bigcup_{i=t-\epsilon}^{t-1} r_{i+1}(t+1)$. Now, we can compare $R(t)$ with $R(t+1)$ term by term. That is, we can compare $r_i(t)$ from $R(t)$ with $r_{i+1}(t+1)$ from $R(t+1)$. By Corollary 12, we know that cardinality of both terms are equal for $t - \epsilon \leq i \leq t-1$. That last thing to show is that stochastic process gives us equality. Consider the following random process, there are n coupons. We can draw coupon ϵ trials by the following rule. For i-th trial, we draw $r_i(t)$ distinct number of coupons randomly. Let X be a random variable representing number of distinct coupons collected for ϵ trials. In this situation, $R(t)$ and $R(t+1)$ both represent X. Therefore, the expectation of two random variables must be equal because $R(t)$ and $R(t+1)$ are random variables of identical stochastic process.

Hence, $E[|R(t+1)|] = E[|R(t)|]$ for $t \geq \epsilon$. That is, $y_\epsilon(t) = y(\epsilon)$ for $t \geq \epsilon$. ◀

This result implies that we can use t and ϵ interchangeably since the *hvc* size of ϵ-constrained time model reaches equilibrium point after $t \geq \epsilon$.

6 Simulation Results

In this section, we evaluate our analytical model by comparing to simulation results. Since the analytical results in this paper are captured by Theorems 3–13, we perform simulation experiments to validate them.

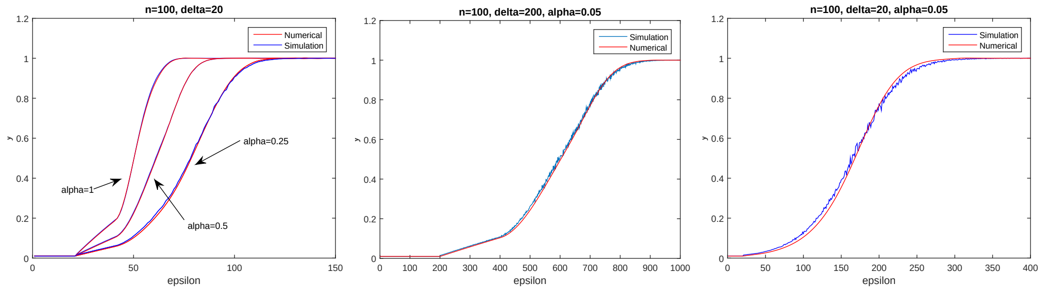


Figure 2 Simulation vs. Numerical Results from Theorem 3.

Simulation Setup. We implement according to our model in various configurations. For the purpose of experiments, we simulate distributed processes with central absolute time in one machine. We simulate sending event by adding a new message to priority queue of destination process with on arrival time $t + \delta$ in future. At each absolute time, each process checks if its inbox has messages with deliver time less than or equal to t. If so, we perform receive events. We repeat until no such message exists in the inbox. Each process has an access to physical time with ϵ-uncertainty interval guarantee. While we simulate sending and receiving events using central absolute time, the absolute time is oblivious to the processes. For purpose of reproducibility, all source codes for simulation are available at `http://www.cse.msu.edu/~yingchar/hvc.html`. All parameters are configurable.

6.1 Analytical vs. Simulation Results (Validation of Theorem 3)

In this case, we compare the analytical model from Theorem 3 with simulation results. For analytical solution, we use standard numerical solver *dde*23 in MATLAB. For experiments, we run for sufficiently long time so that the active clock (i.e., number of *hvc* entries) is stabilized. In particular, we plot the result for ϵ from various value of ϵ. For each ϵ, we run simulation for t up to 2000 and calculate the average starting from $t = \epsilon$ since we start from a state where a process knows only its own clock, we omit the initial clock values where the size of the clock is small.

The results are shown in Figure 2. We overlay numerical solution and simulation results. In Figure 2 (left), we set $n = 100, \delta = 20$ and run for three different values of $\alpha = 1$, 0.5 and 0.25 respectively. We overlay numerical solution and simulation results. In Figure 2 (mid) and (right), we set $n = 100, \alpha = 0.05$ and different values of $\delta = 200$ and 20, respectively. Note that the middle figure has $\alpha * \delta = 10$ where as the right figure has $\alpha * \delta = 1$. We notice the difference of $\alpha * \delta$ and its effect to characteristic of the plot. When $\alpha * \delta$ is small as suggested by Theorem 5, the graph looks like sigmoid function. This shows our analytical model in Theorem 3 gives us an exact plot with simlation results with minimal error. We notice slight perturbation for small value of α. This is due to discontinuity of the discrete events.

From these results, we corroborate that the relation predicated in Theorem 3 is valid and tight.

Given that we have an exact analytical model, we now consider the bound we have for closed form approximation results. From now on, we use the numerical solution as a baseline.

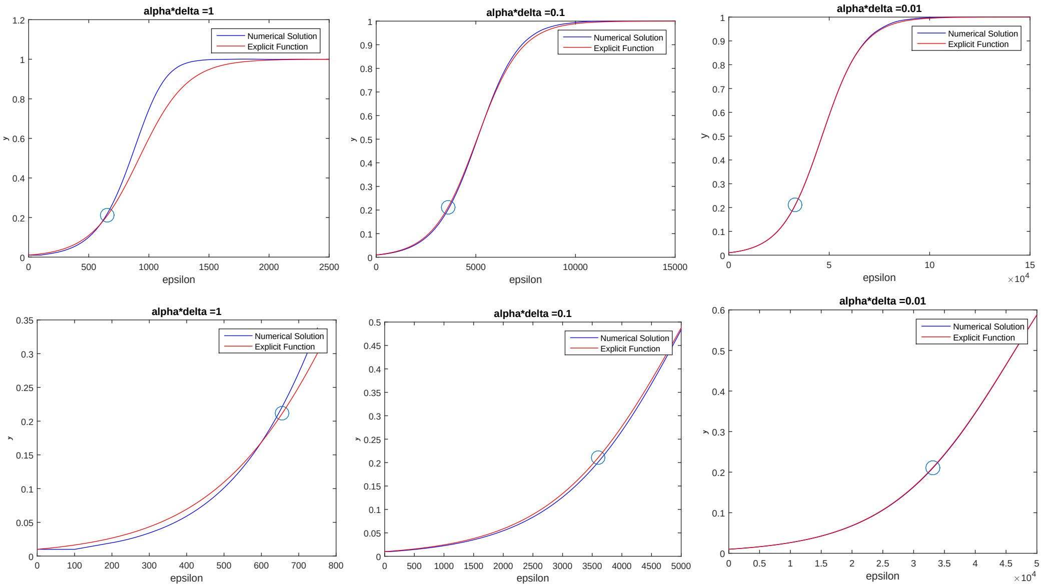


Figure 3 We compare explicit function with numerical solution. The circle is the phase transition ϵ_p obtained by Theorem 6. The bottom figures are zoomed version of the corresponding upper ones.

6.2 Explicit Form vs. Numerical Solutions (Validation of Theorem 5, 6 and 8)

Theorem 5 gives an explicit function when $\delta * \alpha$ is arbitrarily small. How small does it need to be is a subject of this section. In Figure 3, we fix $\delta = 100$ and vary α for $\delta * \alpha = 1, 0.1$,and 0.01, respectively. This shows that the explicit function in Theorem 5 is identical to numerical solution when $\alpha * \delta$ is small. Typical value is $\alpha * \delta < 1$. Note that phase transition $\epsilon_p = (\alpha^{-1} + \delta)(\ln((2 - \sqrt{3})(n-1)))$ is shown in circle. The bottom figures are the zoomed-in version of corresponding top ones.

If $\alpha * \delta$ is large, the hvc size is typically big during $t < 3\delta$. We obtain the upper and lower bound close forms using a technique called method of steps in delay differential equation. We evaluate the result accordingly.

By Theorem 8, we have exact solution for $t \leq 2\delta$ and approximate closed form for $t \in [2\delta, 3\delta]$. We simulate in various configurations. The result is shown in Figure 4. According to these experiments, Theorem 8 gives us an exact bound during $t \in [\delta, 2\delta]$ and reasonable upper and lower bound during $t \in [2\delta, 3\delta]$. Note that when α is small, the approximation converges to exact as shown in Figure 4 (mid and right).

6.3 Unconstrained vs. ϵ-constrained Time (Validation of Theorem 13)

We evaluate relationship between $y(t)$ (from the unconstrained time model) and $y_\epsilon(t)$ (from ϵ-constrained time model). Theorem 13 implies that $y_\epsilon(t) = y(t)$ for $t \geq \epsilon$. Specifically, in Figure 5, we simulate the programs for 100 processes with $\alpha = 0.25$ and $\delta = 10$. The result show that $y_x(t)$ is almost same as $y(t)$ when $t \leq x$. And, $y_x(t)$ is almost same as $y(x)$ when $t > x$ for $x = 30, 60$. This conforms to the prediction in Theorem 13.

In addition, we plot the distribution of sizes of hvc of all processes at each time. In Figure 5 (right), we plot box distribution which is based on normal distribution. The middle point represents average value at time t. The thick area represents area within a standard

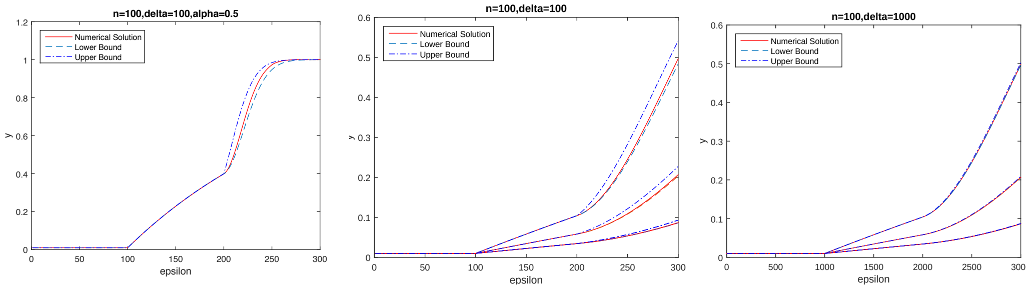


Figure 4 Numerical solution vs. closed form for $t \le 3\delta$. Note we use $\alpha = 0.1, 0.05$ and 0.025 for middle figure. For right figure, we use $\alpha = 0.01, 0.005$, and 0.0025, respectively.

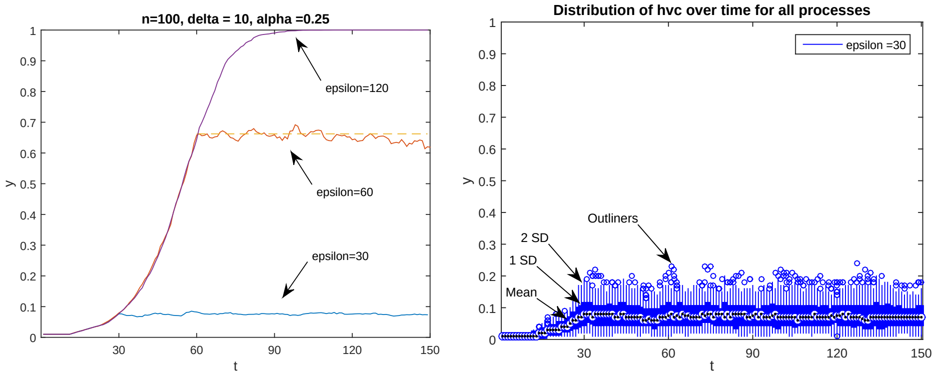


Figure 5 Validation of Theorem 13. (Left) We plot three graphs of $n = 100, \delta = 10$, and $\alpha = 0.25$. Each graph uses different value of ϵ. Note that after $t = \epsilon$, the function is stabilized. (Right) We plot actual distribution in terms of box plot for each time t.

deviation. The thin area represents twice standard deviation. The above dots are outliners. During before phase transition, we can expect a distribution around $y(\epsilon)$.

From these results, we find that the relation predicated in Theorem 13 is valid.

7 Practical considerations for HVC sizes and the phase transition

Our analytical derivations and simulation experiments point to a phase transition on HVC size. Here we use typical values for δ and α from datacenter environments, and determine the phase transition threshold. We show that ϵ achieved using NTP is much less than this phase transition threshold, so for practical distributed systems and modern deployments, the HVC sizes will remain very small and significantly less than n.

For convenience, we calculate phase transition ϵ^* in terms of seconds rather than unit time (clock tick) as follows. Let r be resolution of the time protocol, e.g., NTP has a theoretical resolution of 2^{-32} seconds (233 picoseconds) [15]. The unit of r is seconds per clock tick. Define f as messages frequency in terms of number of messages per second. Also, let d be minimum messages delay in terms of seconds. It is easy to see that $\alpha = fr$ and $\delta = \frac{d}{r}$. We assume that α is small. This is true when the time protocol has sufficient resolution.

[2] Technically, number of messages are equivalent to number of clock ticks that trigger the sending events. Hence, the message rate α means proportion of such clock ticks over all clock ticks in the long run.

If $\alpha * \delta << 1$, or equivalently $f * d << 1$, we can apply these values into our phase transition formula in Theorem 6 to obtain $\epsilon_p = \frac{(f^{-1}+d)}{r} \ln((2-\sqrt{3})(n-1))$. Note that ϵ_p has unit of clock ticks. We can convert to seconds by multiplying by r. Therefore, $\epsilon^* = (f^{-1}+d)\ln((2-\sqrt{3})(n-1))$. In other words, ϵ^* is proportional to $f^{-1}+d$, for fixed value of n.

For example, consider the following configuration. A small value for d is 1 millisecond. To determine a value for α, communication frequency, we will consider chatty nodes that send 100 messages a second to other nodes in a distributed system of n nodes. Most practical applications in fact use orders of magnitude lower α, since reducing message communication rates can improve efficiency of distributed systems. Techniques for reducing α include aggregation and batching of messages before sending a message.

Since $\alpha * \delta << 1$, we can use Theorem 6 to calculate the phase transition threshold ϵ^*. When we substitue above values for d and f with $n = 100$ in the formula, we get $(100^{-1}+10^{-3})\ln((2-\sqrt{3})(99)) = 0.036$ seconds.

In this situation, it is possible to get ϵ less than 10ms using NTP synchronized clocks, which is less than the phase transition for above configuration. Our simulation results show that this corresponds to an average 1.1 *hvc* size for 100 nodes. That is each *hvc* clock maintains only few entries most of the time, yet when it is needed *hvc* expands on demand to allow more entries to capture causality both ways in the ϵ uncertainty slices. Therefore, we can see that for various deployments of practical distributed applications, our HVC component will avoid the phase transition and achieve very small *hvc* sizes. Moreover, using less chatty nodes, hence a smaller α communication frequency, would also lead to larger ϵ^*.

Finally, $\epsilon_p = (\frac{1}{\alpha}+\delta)(\ln((2-\sqrt{3})(n-1)))$ also tells us that the HVC sizes scale very well with respect to n, the number of nodes in the network. As we can see from the above equation increase in n will increase ϵ_p, and will delay the critical phase transition of HVC sizes. In the traditional VC, the size of the VC increase directly with n. In HVC, surprisingly, the increase of n, benefits in delaying the phase transition and reducing the size of HVC. The intuition behind this is that, the critical phase transition occurs when a process not only gets entries added to its clock from direct interaction but also from indirect transfer with another processes' HVC entries. This indirect hearing and addition makes the HVC entries blow up. For larger n, the probability of such indirect addition reduces slightly, and hence larger n, delays the phase transition to large HVC sizes.

7.1 Extensions to the model

Unicast is the predominant communication pattern in cloud computing systems. Sending a message to many recipients is often implemented in terms of multiple unicast messages. Our modeling however failed to capture the incast problem that occurs back when several nodes send message back to the same node. Due to large fan-in/fan-outs in some cloud computing, and especially web services systems, incast problem may occur. When many nodes may be sending back to the same node at the same time, this may grow the number of HVC entries at the recipient beyond what the model predicts.

In addition, our model uses a single worst case ϵ to denote clock synchronization uncertainty in the system. However, in a large scale distributed system, there will some nodes that are more tightly synchronized with real time, versus some nodes that are poorly synchronized with real time. It is possible to go finer grain tracking of ϵ and record and use per-node ϵ. We leave this as future work to explore.

8 Related work

Logical clock (LC) [12] was proposed in 1978 by Lamport as a way of timestamping and ordering events in an asynchronous distributed system. In 1988, the vector clock (VC) [10, 13] was proposed to maintain a vectorized version of LC. VC maintains a vector at each node which tracks the knowledge this node has about the logical clocks of other nodes. While LC finds one consistent snapshot (that with same LC values at all nodes involved), VC finds all possible consistent snapshots, which is useful for debugging applications. Unfortunately, the space requirement of VC is on the order of nodes in the system, and is prohibitive, and it stays prohibitive with optimizations [9, 18, 14] that reduce the size of VC. Resettable vector clocks (RVC) generalizes the notion of VC, and provides a bounded-space and fault-tolerant implementation of VC applications. To this end, RVC identifies an interface contract under which the RVC implementation can be substituted for VC in the client applications, without affecting the client's correctness. The number of entries in RVC is still n, the number of nodes in the system.

A version vector (VV) [16] is a version of VC that is customized for summarizing the set of updates applied to a replica in a replicated database system. While VC is employed for establishing a partial order among a set of events occurring in the nodes, VV is employed for establishing a partial order among the replicas in the distributed system. VV generally uses less number of entries (typical replica sizes are 3 or 5) and offers more opportunities for bounding the size of each entry [1]. As another approach to reducing VV sizes, a unilateral VV pruning algorithm is introduced using loosely synchronized clocks [17]. That algorithm assumes synchronous networking: it demands that each event be delivered to all live nodes and processed by them within a fixed period [17].

The clocks discussed above have been adopted by many cloud computing systems. Dynamo [19] adopts version vectors for causality tracking of updates to the replicas. Orbe [7] uses dependency matrix along with physical clocks to obtain causal consistency. In the worst case, both these solutions require large timestamps. Cassandra uses physical time and LWW-rule for updating replicas. Spanner [5] employs TrueTime (TT) to order distributed transactions at global scale, and facilitate read snapshots across the distributed database. TT relies on a well engineered tight clock synchronization available at all nodes thanks to GPS clocks and atomic clocks made available at each cluster. In order to ensure $e\ \underline{hb}\ f \Rightarrow tt.e < tt.f$ and provide consistent snapshots, Spanner requires waiting-out uncertainty intervals of TT at the transaction commit time which restricts throughput on writes. In contrast, HVC and HLC does not require waiting out the clock uncertainty, since they are able to record causality relations within this uncertainty interval using the VC and LC update rules.

A recent work [2] surveys the use of clocks in cloud computing and investigates how the logical and physical clock concepts are applied in the context of developing distributed data store systems for the cloud and review the choice of clocks in relation to consistency/performance tradeoffs.

An alternate approach for ordering events is to establish explicit relation between events. This approach is exemplified in the Kronos system [8], where each event of interest is registered with the Kronos service, and the application explicitly identifies events that are of interest from causality perspective. This allows one to capture causality that is application-dependent at the increased cost of searching the event dependency relation graph.

9 Conclusion

We presented an analytical model to compute the size of HVC. This analytical model had four parameters: ϵ (window of uncertainty), δ (minimum message delay), α (message rate) and n (size of the network). We presented a differential equation whose solution provides the estimated size of HVC. We also identified closed form solutions for some special cases. We used simulation results to validate the analytical model. In particular, we showed that the results predicated by the analytical model are identical to the simulation results. Moreover, the upper and lower bounds computed by the closed form solutions are also very close to the simulation results. Hence, they can be used to predict the size of HVC in the given system setting. We also showed that many deployments of practical distributed applications will avoid the phase transition easily and achieve very small HVC sizes, significantly less than n, the number of nodes in the system.

References

1 J. Almeida, P. Almeida S. Paulo, and C. Baquero. Bounded version vectors. *Distributed Computing: 18th International Conference, DISC 2004*, pages 102–116, 2004.

2 M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull*, 38(1):18–31, 2015.

3 B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

4 Cockroachdb: A scalable, transactional, geo-replicated data store. http://cockroachdb.org/.

5 J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. *Proceedings of OSDI*, 2012.

6 M. Demirbas and S. Kulkarni. Beyond truetime: Using augmentedtime for improving google spanner. *LADIS'13: 7th Workshop on Large-Scale Distributed Systems and Middleware*, 2013.

7 J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC'13, pages 11:1–11:14, New York, NY, USA, 2013. ACM. `doi:10.1145/2523616.2523628`.

8 R. Escriva, A. Dubey, B. Wong, and E.G. Sirer. Kronos: The design and implementation of an event ordering service. *EuroSys*, 2014.

9 M. Ahamad F. J. Torres-Rojas. Plausible clocks: Constant size logical clocks for distributed systems. *Proceedings of WDAG*, pages 71–88, 1996.

10 J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.

11 S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Principles of Distributed Systems*, pages 17–32. Springer, 2014.

12 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

13 F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

14 S. Meldal, S. Sankar, and J. Vera. Exploiting locality in maintaining potential causality. In *Proceedings of Principles of Distributed Computing (PODC)*, pages 231–239, 1991. doi:10.1145/112600.112620.

15 D. Mills. A brief history of ntp time: Memoirs of an internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33(2):9–21, 2003.

16 D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983. doi:10.1109/TSE.1983.236733.

17 Y. Saito. Unilateral version vector pruning using loosely synchronized clocks. Technical report, HP Labs, 2002.

18 M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, 1992.

19 W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

Non-Blocking Doubly-Linked Lists with Good Amortized Complexity

Niloufar Shafiei

Department of Electrical Engineering and Computer Science, York University,
4700 Keele Street, Toronto, Ontario, Canada
niloo@cse.yorku.ca

Abstract

We present a new non-blocking doubly-linked list implementation for an asynchronous shared-memory system. It is the first such implementation for which an upper bound on amortized time complexity has been proved. In our implementation, operations access the list via *cursors*. Each cursor is located at an item in the list and is local to a process. In our implementation, cursors can be used to traverse and update the list, even as concurrent operations modify the list. The implementation supports two update operations, insertBefore and delete, and two move operations, moveRight and moveLeft. An insertBefore(c, x) operation inserts an item x into the list immediately before the cursor c's location. A delete(c) operation removes the item at the cursor c's location and sets the cursor to the next item in the list. The move operations move the cursor one position to the right or left. Update operations use single-word Compare&Swap instructions. Move operations only read shared memory and never change the state of the data structure. If all update operations modify different parts of the list, they run completely concurrently. A cursor is active if it is initialized, but not yet removed from the process's set of cursors. Let $\dot{c}(op)$ be the maximum number of active cursors at any one time during the operation *op*. The amortized step complexity is $O(\dot{c}(op))$ for each update *op* and $O(1)$ for each move. We provide a detailed correctness proof and amortized analysis of our implementation.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases non-blocking data structure, doubly-linked list, shared memory, amortized complexity, cursor

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.35

1 Introduction

The linked list is a fundamental data structure that has many applications in distributed systems including processor scheduling, memory management and sparse matrix computations [9, 15, 17]. It is also used as a building block for more complicated data structures such as deques, skip graphs and Fibonacci heaps. We design a concurrent doubly-linked list for asynchronous shared-memory systems that is *non-blocking* (also sometimes called *lock-free*): it guarantees some operation will complete in a finite number of steps. Our list has a full proof of correctness and analysis of its amortized complexity. The first non-blocking *singly*-linked list [23] was proposed two decades ago. Designing a non-blocking *doubly*-linked list was an open problem for a long time. Doubly-linked lists have been implemented using multi-word synchronization primitives that are not widely available [1, 10]. Sundell and Tsigas [21] gave the first implementation from single-word compare&swap (CAS). However, they did not provide a correctness proof and their implementation has some problems. (See Section 2.)

A process accesses our list via a *cursor*, which is an object in the process's local memory that is located at an item in the list. Update operations can insert or delete an item at the

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 35; pp. 35:1–35:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

cursor's location, and moveLeft and moveRight operations move the cursor to the adjacent item in either direction. If the item where a cursor c located at is removed by another process, an operation called with c first needs to *recover c's location* in the list. In our list, recovering a cursor's location and moving a cursor are achieved using only reads of shared memory, even when there are concurrent updates. Thus, in our list, only updates might interfere with other concurrent operations. However, if all concurrent updates are on disjoint parts of the list, they do not interfere with one another. Our implementation is modular and can be adapted for other updates, such as replacing one item by another. For simplicity, we assume the existence of a garbage collector such as the one in Java.

In Section 3, we give a novel specification that describes how updates affect cursors and how a process gets feedback about other processes' updates at the location of its cursor. This interface makes the list easy to use as a black box. In our implementation, a cursor c becomes *invalid* if an update is performed at c's location using another cursor. If an operation is called with an invalid cursor, it returns invalidCursor and makes the cursor valid again. This avoids having a process perform an operation on the wrong item. If an insertion is performed before a cursor c using another cursor, c becomes invalid for insertions only, to ensure that an item can be inserted between two specific items. This makes it easy to maintain a sorted list. For example, if two processes try to insert 5 and 7 at the same location simultaneously, one fails and returns invalidCursor. This avoids inserting 7 and then 5 out of order.

We provide a detailed proof that our list is *linearizable*: each operation appears to take place atomically at some time during the operation. One of the main challenges is to ensure the two pointer changes required by an update appear to occur atomically. Our implementation uses two CAS steps to change the pointers. Each update appears to take effect at the first CAS. Between the two CAS steps, the data structure is temporarily inconsistent. We design a mechanism for detecting such inconsistencies by only reading the shared memory, which allows concurrent operations to behave as if the second change has already occurred. This makes moves and recovering cursors' locations very efficient.

We give an amortized analysis of our implementation (excluding garbage collection), which is the first for any non-blocking doubly-linked list. A cursor is active if it is initialized, but not removed from the process's set of cursors. Let $\dot{c}(op)$ be the maximum number of active cursors at any one time during the operation op. The amortized complexity of each operation op is $O(\dot{c}(op))$ for updates and $O(1)$ for moves. Due to space restrictions, we sketch the proof of correctness and amortized analysis here. Complete details are in [19].

To summarize the contributions of this paper:

- We present a non-blocking linearizable doubly-linked list using single-word CAS.
- The cursors provided by our implementation are robust: they can be used to traverse and update the list, even as concurrent operations modify the list.
- Cursors' locations are recovered and cursors are moved by only reading the shared memory.
- Our implementation and proof are modular and can be adapted for other data structures.
- Our implementation can easily maintain a sorted list.
- In our list, the amortized complexity of each update op is $O(\dot{c}(op))$ and each move is $O(1)$.
- We empirically show our list outperforms the one in [21] on a multi-core machine.

2 Related Work

In this paper, we focus on non-blocking algorithms of doubly-linked lists, which do not use locks. There are two general techniques for obtaining non-blocking data structures: universal constructions (see [7] for a survey) and transactional memory (see [11] for a survey). Such

Implementation	supports cursors?	operations to move cursor?	recover cursor's location?	primitive used	# of CAS with no contention
Greenwald [10]	No	-	-	2-CAS	depends on size of list
Attiya and Hillel [1]	Yes	No	No	2-CAS	13-15
Sundell and Tsigas [21]	Yes	Yes (CAS used)	Yes (CAS used)	CAS	2-4
List presented here	Yes	Yes (No CAS)	Yes (No CAS)	CAS	5

Figure 1 Implementations of doubly-linked lists.

general techniques are usually less efficient than implementations designed for specific data structures. Turek, Shasha and Prakash [22] and Barnes [2] introduced a technique in which processes cooperate to complete operations to ensure non-blocking progress. Each update operation stores information that other processes can use to help complete the update in a descriptor object. This technique has been used for various data structures. Here, we extend the scheme used in [3, 4, 5, 8] to coordinate processes for tree structures and the scheme used in [18] for updates that make more than one change to a Patricia trie. The implementations in [3, 4, 5, 8] only handle one change atomically, but updates in our list make multiple changes atomically using a simpler scheme than the one used in [18]. However, handling the cursors and move operations in our list are original.

The k-CAS primitive modifies k locations atomically. Although it is usually not available in hardware, it can be implemented from single-word CAS [12, 16, 20]. Doubly-linked lists can be implemented using k-CAS, but it is not completely straightforward to do so. Suppose each item is represented by a node with nxt and prv fields that point to the adjacent nodes. Consider a list of four nodes, A, B, C and D. A deletion of C must change the nxt pointer in B from C to D and the prv pointer in D from C to B. It is *not* sufficient for the deletion to update these two pointers with a 2-CAS. If two concurrent deletions remove B and C in this way, C would still be accessible through A after the deletions. This problem can be avoided if the deletion of C uses a 4-CAS to simultaneously update the two pointers and check whether the two pointers of C still point to B and D. Then, the 4-CAS of one of the two concurrent deletions would fail. The most efficient k-CAS implementation [20] uses $2k + 1$ CAS steps to update k words when there is no contention. Thus, at least 9 CAS steps would be used for a 4-CAS. Moreover, although the 4-CAS works for updating pointers, it is not obvious how to recover a cursor's location when another process deletes the cursor's node.

Greenwald's doubly-linked list [10] uses 2-CAS, but does not provide cursors. His approach does not support concurrency: all processes cooperate to execute one operation at a time.

Attiya and Hillel [1] proposed a doubly-linked list using 2-CAS, but it only supports update operations. It has the nice property that concurrent updates can interfere with one another only if they are changing nodes close to each other. If there is no interference, an update performs 13 to 15 CAS steps (and one 2-CAS). Their implementation does not recover a cursor's location, so deletions might make other processes lose their place in the list. They also give a restricted implementation using single-word CAS, in which deletions can be performed only at the ends of the list.

None of the implementations that use k-CAS handle cursors with the same functionality as ours. (See Figure 1.) Since the implementations in [1, 10] do not provide a way to traverse the list, they are not complete implementations of a doubly-linked list. Moreover, they all perform many CAS steps for contention-free updates, whereas ours performs only five.

Sundell and Tsigas [21] gave the first non-blocking doubly-linked list using single-word CAS (although a word must store both a bit and a pointer). Non-blocking data structures are notoriously difficult to design, so detailed correctness proofs are essential. In [21], the claim of linearizability is justified by defining linearization points without proving that they are correct. The implementation has at least minor errors: using the Java PathFinder model checker [13], we found an execution that incorrectly dereferences a null pointer. We contacted the authors, who suggested changing some lines to fix the problem, but a correctness proof of the revised version is still lacking. Their implementation is ingenious but quite complicated. In particular, the helping mechanism is very complex, partly because updates can terminate before completing the necessary changes to the list, so operations may have to help non-concurrent updates.

There are a number of differences between the designs of our list and the one in [21]. In [21], to recover a cursor's location or to move a cursor, sometimes CAS steps are performed. Our approach is quite different, allowing a cursor's location to be recovered and cursors to be moved only by reading shared memory, even when there is contention. So, moves do not interfere with one another. This is a desirable property since moves are more common than updates in many applications. In the best case, the updates in [21] perform 2 to 4 CAS steps. However, moves must perform CAS steps to help complete updates. In fact, deletions can construct long chains of deleted nodes whose pointers do not get updated by the deletions. Then, a move may have to traverse this chain, performing CAS steps at every node. In our list, an update helps only updates that are concurrent with itself, and moves do not help at all. Our implementation can easily be used to maintain a sorted list. It is not straightforward to see how the implementation in [21] could maintain a sorted list. Our empirical evaluation shows our list outperforms the one in [21] on a multi-core machine.

3 The Sequential Specification

We give a sequential specification, which describes how operations behave when performed one at a time. (A more formal specification is available in [19].) This specification is extended to concurrent implementations by requiring them to be linearizable [14].

A list is a pair (L, S) where L is a finite sequence of items ending with a special end-of-list marker (EOL), and S is a set of cursors. Each cursor c in S is located at one item in L and $c.item$ presents that item. Eight types of operations are supported: initializeCursor, destroyCursor, resetCursor, moveRight, moveLeft, get, delete and insertBefore. An **initializeCursor**(c) adds new cursor c to S whose item is the first item in L. A **destroyCursor**(c) removes cursor c from S. A process p can call an operation with a cursor c only if p itself initialized c and c has not been removed from S. A **resetCursor**(c) locates c at the first item in L. A **moveRight**(c) advances c's location to the next item in L and returns true, (unless it is already at the last item in L, in which case it returns false and has no effect). Similarly, **moveLeft**(c) sets c's location to the previous item in L and returns true, (unless it is already at the first item in L, in which case it returns false and has no effect). A **get**(c) returns the value of $c.item$.

To keep track of cursor invalidation, each cursor in S has two additional fields called $invDel$ and $invIns$, which indicates whether the cursor is invalid for different operations. A **delete**(c) removes the item that c is located at and returns true, (unless c is located at EOL in which case the delete returns false). If the deletion is successful, it also moves each cursor c' located at the deleted item to the next item in L. This ensures that all cursors continue to point to items that are currently in the list, so that no other process can lose its place in the list as a result of the deletion. For each cursor $c' \neq c$ that is moved as a result of the delete, $c'.invDel$ is set to true so that if the next operation called with c' is an update, get or move, it returns invalidCursor to indicate that c' has been moved. This ensures that the operation is not inadvertently applied to the wrong location. For example, suppose a process p performs a delete(c) that removes item x. If another process p' has cursor c' located at x, $c'.item$ is set to the next item y in L and c' becomes invalid (i.e., $c'.invDel$ becomes true). Thus, the deletion cannot cause c' to lose its place in L. Since x is removed by p, p' does not yet know that c' is no longer located at x. If p' then calls a delete(c') to attempt to remove x, since $c'.invDel$ is true, it returns invalidCursor and does not remove y. When $c'.invDel$ is true, the next operation called with c' sets $c'.invDel$ to false, making c' valid again.

An **insertBefore**(c, v) adds a new item with value v to the left of c's location and returns true. When an insertBefore(c, v) succeeds, each other cursor c' located at the same item as c becomes invalid for insertions only. This is indicated by setting $c'.invIns$ to true. If an insertBefore(c', v') operation is called when $c'.invIns$ is true, it returns invalidCursor. This invalidation ensures an item can be inserted between two specific items in the list. For example, suppose we wish to maintain L so that values of items are sorted and process p' has a cursor c' whose item's value is 5. Then, p' advances c' to the next item in the sequence, which has value 8. If 7 is inserted before 8 by another process p, c' becomes invalid only for insertions (i.e., $c'.invIns$ becomes true). Since 7 is inserted by p, p' does not yet know that the item before 8 is 7. If p' then calls an insertBefore operation with c' to attempt to insert 6 before 8, it does not succeed because that would place 6 between 7 and 8. Since $c'.invIns$ is true, insertion by p' returns invalidCursor instead. When $c'.invIns$ is true, the next operation called with c' sets $c'.invIns$ to false, making c' valid for insertions again.

4 The Non-blocking Implementation

List items are represented by Node objects, which have pointers to adjacent Nodes. A Cursor object is simply a pointer to a Node in a process's local memory. Updates are done in several steps as shown in Fig. 2 and 3. To avoid simultaneous updates to overlapping parts of the list, an update flags a Node before removing it or changing one of its pointers. This flag acts like a lock on the Node's pointers. To ensure the non-blocking property, other operations can help complete the update that placed the flag and then remove the flag. To facilitate this, a Node is flagged by storing a pointer to an Info object, which is a descriptor of the update and contains the information needed to help complete the update. List pointers are updated using CAS, so that helpers cannot perform an operation more than once.

The correctness of algorithms using CAS often depends on the fact that, if a CAS on variable V succeeds, V has not changed since an earlier read. An ABA problem occurs when V changes from one value to another and back before the CAS occurs, causing the CAS to succeed when it should not. When a Node *new* is inserted between Node x and y, we replace y by a new copy, *yCopy* (Fig. 3). This avoids an ABA problem that would occur if, instead, insertBefore simply changed the pointers in x and y to *new*, because a subsequent deletion of *new* could then change x's pointer back to y again. Creating a new copy of y also makes invalidation of Cursors for insertions easy. An insertion of a Node before y writes a permanent pointer in y to *yCopy* before replacing y, so that any other process whose Cursor is at y can detect that an insertion has occurred there and update its Cursor to *yCopy*.

The objects used in our implementation are described in line 1 to 16 of Fig. 4. A Node has the following fields. The *val* field contains the item's value, *nxt* and *prv* point to the next and previous Nodes in the list, *copy* points to a new copy of the Node (if any), *info* points to an Info object that is the descriptor of the update that last flagged the Node, and *state* is initially ordinary and is set to copied (before the Node is replaced by a new copy)

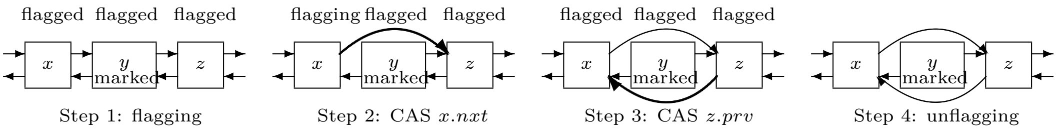


Figure 2 A delete operation.

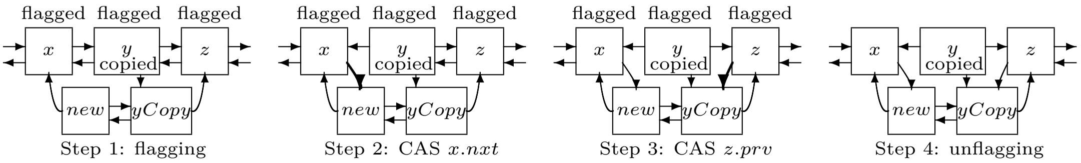


Figure 3 An insertBefore operation.

or marked (before the Node is removed). The *info* field is initially set to a dummy Info object, *dum*. The *info*, *nxt* and *prv* fields of a Node are changed using CAS steps. We call the steps that try to modify these three fields *flag CAS*, *forward CAS* and *backward CAS* steps, respectively. To avoid special cases, we add sentinel Nodes *head* and *tail*, which do not contain values, at the ends of the list. They are never changed and Cursors never move to *head* or *tail*. The last Node before *tail* always contains the value EOL.

Info objects are used as our update operation descriptors. An Info object I has the following fields. $I.nodes[0..2]$ stores the three Nodes x, y, z to be flagged before changing the list. $I.oldInfo[0..2]$ stores the expected values to be used by the flag CAS steps on x, y and z. $I.newNxt$ and $I.newPrv$ store the new values for the forward and backward CAS steps on $x.nxt$ and $z.prv$. $I.rmv$ indicates whether y should be removed from the list or replaced by a new copy. $I.status$, indicates whether the update is inProgress (the initial value), committed (after the update is completed) or aborted (after a Node is not flagged successfully). (One exception is the dummy Info object *dum* whose *status* is initially aborted.) The *status* field is the only field of I whose value might be changed after I's creation. A Node is *flagged for* I if its *info* field is I and $I.status =$ inProgress. Thus, setting $I.status$ to committed or aborted also has the effect of removing I's flags. As with locks, successful flagging of the three Nodes guarantees that the operation will be completed successfully without interference from other operations. Unlike locks, if the process performing an update crashes after flagging, other processes may complete its update using the information in I.

4.1 Detailed Description of the Algorithms

Pseudo-code for our implementation is given in Fig. 4.

We say a Node v is *reachable* if there is a path of *nxt* pointers from *head* to v. At all times, the reachable Nodes correspond to the items in the list. So, each update is linearized when its forward CAS succeeds (step 2 of Fig. 2 and 3). Just after this CAS, y becomes unreachable. We prove that no process changes $y.nxt$ or $y.prv$ after that, so $y.prv$ remains equal to x. Since there is no ABA problem, $x.nxt$ is never set back to y after y becomes unreachable. Thus, the test $y.prv.nxt \neq y$ tells us whether y has become unreachable (even between the successful forward and backward CAS of the update that removed y). This test is used in recovering the location of a Cursor whose *node* is removed from the list (line 77) and also by moveLeft operations (line 47).

Since a Cursor c is a pointer in a process's local memory, it becomes out of date if the Node it points to is deleted or replaced by another process's update. Thus, at the beginning of

```
type Cursor
  Node $node$                                    ▷ location of Cursor

type Node                                        ▷ represent list item
  Value $val$
  Node $nxt$                                     ▷ next Node
  Node $prv$                                     ▷ previous Node
  Node $copy$                    ▷ new copy of Node (if replaced)
  Info $info$                              ▷ descriptor of update
  {copied, marked, ordinary} $state$ ▷ indicates if Node
    is copied by an insertion or marked for deletion

type Info                              ▷ Descriptor of an update
  Node[3] $nodes$                          ▷ Nodes to be flagged
  Info[3] $oldInfo$ ▷ expected values of CASs that flag
  Node $newNxt$                      ▷ set $nodes[0].nxt$ to this
  Node $newPrv$                      ▷ set $nodes[2].prv$ to this
  Boolean $rmv$                 ▷ is $I.nodes[1]$ being deleted?
  {inProgress, committed, aborted} $status$

insertBefore($c$: Cursor, $v$: Value):{true, invalidCursor}
  while (true){
    $\langle y, yInfo, z, x, invDel, invIns\rangle \leftarrow$
      updateCursor($c$)              ▷ recover $c$'s location
    if $invDel$ or $invIns$ then return invalidCursor
    $nodes \leftarrow [x, y, z]$
    $oldI \leftarrow [x.info, yInfo, z.info]$
    if checkInfo($nodes$, $oldI$) then{ ▷ no interference
      $new \leftarrow$ new Node($v$, null, $x$, null, $dum$, ordinary)
      $yCopy \leftarrow$ new Node($y.val$, $z$, $new$, null, $dum$,
        ordinary)
      $new.nxt \leftarrow yCopy$
      $I \leftarrow$ new Info($nodes$, $oldI$, $new$, $yCopy$, false,
        inProgress)                        ▷ create descriptor
      if help($I$) then{                  ▷ if insert completed
        $c.node \leftarrow yCopy$           ▷ move $c$ to new copy
        return true }}}

delete($c$: Cursor):{true, false, invalidCursor}
  while (true){
    $\langle y, yInfo, z, x, invDel, \text{-}\rangle \leftarrow$ updateCursor($c$)
                                     ▷ recover $c$'s location
    if $invDel$ then return invalidCursor
    $nodes \leftarrow [x, y, z]$
    $oldI \leftarrow [x.info, yInfo, z.info]$
    if checkInfo($nodes$, $oldI$) then{ ▷ no interference
      if $y.val$ = EOL then return false ▷ $c$ is at last item
      $I \leftarrow$ new Info($nodes$, $oldI$, $z$, $x$, true, inProgress)
                                         ▷ create descriptor
      if help($I$) then{                  ▷ if delete completed
        $c.node \leftarrow z$                ▷ move $c$ to next item
        return true}}}

moveLeft($c$: Cursor):{true, false, invalidCursor}
  $\langle y, -, -, x, invDel, \text{-}\rangle \leftarrow$ updateCursor($c$)
                                     ▷ recover $c$'s location
  if $invDel$ then return invalidCursor
  if $x = head$ then return false        ▷ $y$ is the 1st item
  if $x.prv.nxt \neq x$ and $x.nxt = y$ then{ ▷ $x$ not in list
    if $x.state$ = copied then                ▷ $x$ is replaced
      $c.node \leftarrow x.copy$           ▷ move $c$ to new copy
    else{                                     ▷ $x$ is deleted
      $w \leftarrow x.prv$                ▷ read the item before $x$
      if $w = head$ then return false ▷ $x$ was the 1st item
      $c.node \leftarrow w$}}       ▷ move $c$ to the item before $x$
  else $c.node \leftarrow x$         ▷ move $c$ to the item before $y$
  return true

moveRight($c$: Cursor):{true, false, invalidCursor}
  $\langle y, -, z, -, invDel, \text{-}\rangle \leftarrow$ updateCursor($c$)
                                     ▷ recover $c$'s location
  if $invDel$ then return invalidCursor
  if $y.val$ = EOL then return false  ▷ $y$ is the last item
  $c.node \leftarrow z$              ▷ move $c$ to the item after $y$
  return true

initializeCursor($c$: Cursor):ack
  $c.node \leftarrow head.nxt$
  return ack

destroyCursor($c$: Cursor):ack
  return ack

resetCursor($c$: Cursor):ack
  $c.node \leftarrow head.nxt$           ▷ move $c$ to the 1st item
  return ack

get($c$: Cursor):{Value, invalidCursor}
  $\langle y, -, -, -, invDel, \text{-}\rangle \leftarrow$ updateCursor($c$)
  if $invDel$ then return invalidCursor
  return $y.val$

updateCursor($c$: Cursor):⟨Node, Info, Node, Node,
    Boolean, Boolean⟩
  $invDel \leftarrow$ false
  $invIns \leftarrow$ false
  while( $c.node.prv.nxt \neq c.node$){ ▷ $c.node$ not in list
    if $c.node.state$ = copied then{      ▷ $c.node$ replaced
      $invIns \leftarrow$ true         ▷ make $c$ invalid for insertion
      $c.node \leftarrow c.node.copy$}       ▷ move $c$ to new copy
    if $c.node.state$ = marked then{        ▷ $c.node$ deleted
      $invDel \leftarrow$ true          ▷ make $c$ invalid for deletion
      $c.node \leftarrow c.node.nxt$}}  ▷ move $c$ to the next item
  $info \leftarrow c.node.info$         ▷ read *info* before pointers
  return $\langle c.node, info, c.node.nxt, c.node.prv, invDel,$
    $invIns\rangle$

checkInfo($nodes$: Node[3], $oldInfo$: Info[3]):Boolean
  for $i \leftarrow 0$ to 2{    ▷ detect other updates in progress
    if $oldInfo[i].status$ = inProgress then{
      help($oldInfo[i]$)                    ▷ help other update
      return false}}                        ▷ retry my update
  for $i \leftarrow 0$ to 2                     ▷ detect removed nodes
    if $nodes[i].state \neq$ ordinary then return false
  for $i \leftarrow 1$ to 2           ▷ if flag of 2nd or 3rd node fail
    if $nodes[i].info \neq oldInfo[i]$ then return false
  return true                         ▷ no interference detected

help($I$: Info):Boolean
  $doPtrCAS \leftarrow$ true, $i \leftarrow 0$
  while ($i < 3$ and $doPtrCAS$){
    CAS($I.nodes[i].info$, $I.oldInfo[i]$, $I$) ▷ flag CAS
    $doPtrCAS \leftarrow (I.nodes[i].info = I)$
    $i \leftarrow i + 1$}
  if $doPtrCAS$ then{                    ▷ flag CASs succeeded
    if $I.rmv$ then $I.nodes[1].state \leftarrow$ marked
    else{                                  ▷ in case of insertion
      $I.nodes[1].copy \leftarrow I.newPrv$         ▷ set new copy
      $I.nodes[1].state \leftarrow$ copied}
    CAS($I.nodes[0].nxt$, $I.nodes[1]$, $I.newNxt$)
                                              ▷ forward CAS
    CAS($I.nodes[2].prv$, $I.nodes[1]$, $I.newPrv$)
                                             ▷ backward CAS
    $I.status \leftarrow$ committed} ▷ unflag of successful update
  else if $I.status$ = inProgress then $I.status \leftarrow$ aborted
                       ▷ unflag nodes for unsuccessful update
  return ($I.status$ = committed)
```

Figure 4 Pseudo-code for a non-blocking doubly-linked list.

an update, move or get operation called using c, **updateCursor**(c) is called to bring *c.node* up to date. If *c.node* has been replaced with a new copy by an insertBefore, updateCursor sets *invIns* to true (line 79) and follows the *copy* pointer (line 80). Similarly, if *c.node* has been deleted, updateCursor sets *invDel* to true (line 82) and follows the *nxt* pointer (line 83), which was the next Node at the time of deletion. UpdateCursor repeats the loop at line 77–83 until the test on line 77 indicates that *c.node* is in the list. At the end, updateCursor returns *c.node*, its *info*, *nxt* and *prv* field, *invDel* and *invIns* (line 85).

After calling updateCursor, each update *op* calls **checkInfo** to see if any Node that *op* wants to flag is flagged with an Info object I' of another update. If so, checkInfo calls help(I') (line 89) to try completing the other update, and returns false to indicate *op* should retry. Similarly, if checkInfo sees that another operation has already removed one of the Nodes (line 92) or changed the *info* field of y or z (line 94), it returns false, causing *op* to retry. If checkInfo returns true, *op* creates a new Info object I that describes the update *op* (line 27 or 39) and calls help(I) to try to complete the update (line 28 or 40).

The **help**(I) routine performs the real work of the update. First, it uses flag CAS steps to store I in the *info* fields of the Nodes to be flagged (line 99). If help(I) sees a Node v is not flagged successfully (line 100), help(I) checks if *I.status* is inProgress (line 110). If so, it follows that no helper of I succeeded in flagging all three Nodes; otherwise I's flag on v could not have been removed while I is inProgress. So, v was flagged by a different update before help(I)'s flag CAS. Thus, *I.status* is set to aborted (line 110) and help(I) returns false (line 111), causing *op* to retry.

If the Nodes x, y and z in *I.nodes* are all flagged successfully with I, *y.state* is set to marked (line 103) for a deletion, or copied (line 106) for an insertion. In the latter case, *y.copy* is first set to the new copy (line 105). Then, a forward CAS (line 107) changes *x.nxt* and a backward CAS (line 108) changes *z.prv*. Finally, help(I) sets *I.status* to committed (line 109) and returns true (line 111). A *CAS of I* refers to a CAS step executed inside help(I). We prove below that the *first* forward and *first* backward CAS of I among all calls to help(I) succeed (and no others do).

Both the **insertBefore**(c, v) and **delete**(c) operations have the same structure. They first call updateCursor(c) to bring the Cursor c up to date, and return invalidCursor if this routine indicates c has been invalidated. Then, they call checkInfo to see if there is interference by other updates. If not, they create an Info object I and call help(I) to complete the update. If unsuccessful, they retry.

A **moveRight**(c) calls updateCursor(c) (line 57), which sets *c.node* to a Node y and also returns a Node z read from *y.nxt*. We show there is a time during the move when y is reachable and $y.nxt = z$. If $y.val =$ EOL, the operation returns false (line 59). Otherwise, it sets *c.node* to z (line 60).

A **moveLeft**(c) is more complex because *prv* pointers are updated *after* an update's linearization point, so they are sometimes inconsistent with the true state of the list. A moveLeft first calls updateCursor(c) (line 44), which updates *c.node* to some Node y and also returns a Node x read from *y.prv*. If x is *head*, the operation cannot move c to *head* and returns false (line 46). If the test on line 47 indicates x is reachable, *c.node* is set to x (line 54). This is also done if $x.nxt \neq y$; in this case, we can show that y became unreachable during the move, but *x.nxt* pointed to y just before y became unreachable. Otherwise, x has become unreachable and the test $x.nxt = y$ on line 47 ensures that x was the element before y when it became unreachable. If x was replaced by an insertion, *c.node* is set to that replacement Node (line 49). If x was removed by a deletion, we set *c.node* to *x.prv* (line 53), unless that Node is *head*. We prove in Lemma 12, below, that whenever moveLeft updates *c.node* to some value v, there is a time during the operation when v is reachable and $v.nxt = y$.

5 Correctness Proof

The detailed proof of correctness (available in [19]) is about 50 pages long, so we give only a brief sketch. An execution is a sequence of configurations, $C_0, C_1, \ldots$ such that, for each $i \geq 0$, C_{i+1} follows from C_i by a step of the implementation. For the proof, we assign each Node v a positive real value, called its *abstract value*, denoted $v.absVal$. The $absVal$ of *head*, EOL and *tail* are 0, 1 and 2 respectively. When insertBefore creates the Nodes *new* and *yCopy* (see Fig. 3), $yCopy.absVal = y.absVal$ and $new.absVal = (x.absVal + y.absVal)/2$. The following basic invariant is straightforward to prove.

▶ **Invariant 1.**
- *Any field that is read in the pseudo-code is non-null.*
- *Cursors do not point to* head *or* tail.
- *If $v.nxt = tail$, then $v.val = EOL$.*
- *If $v.nxt = w$ or $w.prv = v$, then $v.absVal < w.absVal$.*

5.1 Part 1: Flagging

Part 1 proves v is flagged for an Info object I when the first forward CAS or first backward CAS of I is applied to Node v. We first show there is no ABA problem on *info* fields.

▶ **Lemma 2.** *The* info *field is never set to a value that has been stored there previously.*

Proof Sketch. The old value used for I's flag CAS on Node v was read from $v.info$ before I is created. So, every time $v.info$ is changed from I' to I, I is a newer Info object than I'. ◀

By Lemma 2, only the first flag CAS of I on each Node in $I.nodes$ can succeed since all such CAS steps use the same expected value. We say I is *successful* if these three first flag CAS steps all succeed.

▶ **Lemma 3.** *After* v.info *is set to I, it remains equal to I until $I.status \neq$ inProgress.*

Proof Sketch. If $v.info$ is changed from I to I', the operation that created I' at line 27 or 39 first called checkInfo on line 23 or 37 and that call returned true. Thus, that call to checkInfo saw $I.status \neq$ inProgress at line 88. ◀

▶ **Observation 4.** *If a process executes line 103–109 inside help(I), I is already successful.*

▶ **Lemma 5.** *If I is successful, $I.status$ is never aborted. Else, $I.status$ is never committed.*

Proof Sketch. If I is not successful, the claim follows from Observation 4. If I is successful, the first flag CAS on each Node in $I.nodes$ succeeds. By Lemma 3, $doPtrCAS$ is not set to false on line 100 until $I.status \neq$ inProgress. So, every call to help(I) evaluates the test on line 102 to true until $I.status \neq$ inProgress. So, no process reaches line 110 before $I.status$ is set to committed on line 109. ◀

▶ **Lemma 6.** *For each of lines 103–109, when the first execution of that line among all calls to help(I) occurs, all Nodes in* I.nodes *are flagged for I.*

Proof Sketch. Suppose one of lines 103–109 is executed inside help(I). By Observation 4, a flag CAS of I already succeeded on each Node in $I.nodes$. By Lemma 5, $I.status$ is never aborted. By Lemma 3, all three Nodes remain flagged for I until some help(I) sets $I.status$ to committed on line 109. ◀

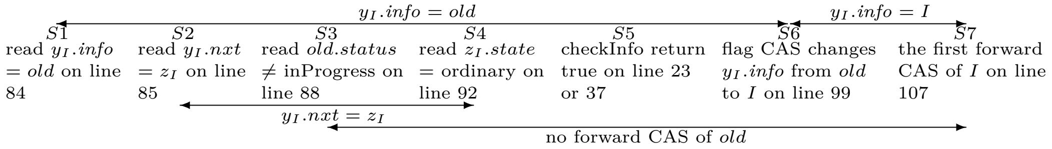

$y_I.info = old$
$y_I.info = I$
$S1$
read $y_I.info$ $= old$ on line 84
$S2$
read $y_I.nxt$ $= z_I$ on line 85
$S3$
read $old.status$ $\neq$ inProgress on line 88
$S4$
read $z_I.state$ $=$ ordinary on line 92
$S5$
checkInfo return true on line 23 or 37
$S6$
flag CAS changes $y_I.info$ from old to I on line 99
$S7$
the first forward CAS of I on line 107
$y_I.nxt = z_I$
no forward CAS of old

Figure 5 Sequence of events used in proof of Lemma 8, Statement 3.

5.2 Part 2: Forward and Backward CAS Steps

Let $\langle y_I, -, z_I, x_I, -, -\rangle$ be the result updateCursor(c) returns on line 19 or 33 before creating I on line 27 or 39. Part 2 of our proof shows that successful flagging ensures that x_I, y_I and z_I are three consecutive Nodes in the list just before the first forward CAS of I, and that the first forward and the first backward CAS of I succeed (and no others do).

▶ **Lemma 7.** *At all configurations after I becomes successful, y_I.info $= I$.*

Proof sketch. To derive a contradiction, assume $y_I.info$ is changed from I to I'. Before creating I', the call to checkInfo returns true, so it sees $I.status \neq$ inProgress at line 88 and then $y_I.state =$ ordinary at line 92. This contradicts the fact that before $I.status$ is set to committed at line 109, $y_I.state$ is set to a non-ordinary value at line 103 or 106 (and no step of the code can change it back to ordinary). ◀

▶ **Lemma 8.**

1. *The first forward and the first backward CAS of I succeed and all other forward and backward CAS steps of I fail.*
2. *The nxt or prv field of a Node is never set to a Node that has been stored there before.*
3. *At the configuration C before the first forward CAS of I, x_I, y_I and z_I are reachable, $x_I.nxt = y_I$, $y_I.prv = x_I$, $y_I.nxt = z_I$ and $z_I.prv = y_I$.*
4. *At all configurations after the first forward CAS of I, $y_I.prv = x_I$ and $y_I.nxt = z_I$.*

Proof sketch. We use induction on the length of the execution.

Statement 1: By induction hypothesis 3, the first forward CAS of I succeeds, since $x_I.nxt = y_I$ just before it. By induction hypothesis 2, no other forward CAS of I succeeds. By induction hypothesis 3, $z_I.prv$ was y_I at some time before the first backward CAS of I. All backward CASes of I use y_I as the expected value of $z_I.prv$, so only the first can succeed (by induction hypothesis 2). By Lemma 6, $z_I.info = I$ at the first forward and first backward CAS of I, and hence at all times between, by Lemma 2. By Lemma 6, no backward CAS of any other Info object changes $z_I.prv$ during this time. So, the first backward CAS of I succeeds.

Statement 2: Intuitively, when the nxt field changes from v to another value, v is thrown away and never used again. (See Fig. 2 and 3). Suppose the first forward CAS of I sets $x_I.nxt$. If I is created by an insertBefore, the CAS sets $x_I.nxt$ to a newly created Node. If I is created by a delete, $z_I.info = I$ at the first forward CAS of I, by Lemma 6. No forward CAS of another Info object I' can change $x_I.nxt$ from z_I to another value earlier, since then $z_I.info$ would have to be I' at the first forward CAS of I, by Lemma 7. The proof for prv fields is symmetric.

Statement 3: First, we prove $y_I.nxt = z_I$ at C. Before I can be created, the sequence of steps $S1, \ldots, S5$ shown in Fig. 5 must occur. By Lemma 6, $y_I.info$ is set to I by some step $S6$ and $y_I.info = I$ at $S7$. By Lemma 2, $y_I.info = old$ between $S1$ and $S6$ and $y_I.info = I$ between $S6$ and $S7$. So, by Lemma 6, only the first forward CAS of old can change $y_I.nxt$

between $S1$ and $S7$. Before $y_I.nxt$ can be changed from z_I to another value by help(old), $z_I.state$ is set to marked or copied (and it can never be changed back to ordinary). So, $y_I.nxt$ is still z_I at $S4$. The first forward CAS of old does not occur after $S3$ since $old.state$ is already committed or aborted at $S3$. So, $y_I.nxt$ is still z_I at C.

By a similar argument, $y_I.prv = x_I$ and x_I, y_I and z_I are reachable in C. The prv and nxt field of two adjacent reachable Nodes might not be consistent at C only if C is between the first forward and first backward CAS of some Info object I' and one of the two Nodes is flagged for I' (step 2 of Fig. 2 and 3). Since x_I, y_I and z_I are flagged for I at C (by Lemma 6), $x_I.nxt = y_I$ and $z_I.prv = y_I$ at C.

Statement 4: By induction hypothesis 3, $y_I.prv = x_I$ at the first forward CAS of I. By Lemma 7, $y_I.info$ is always I after that. So, by Lemma 6, no backward CAS of another Info object changes $y_I.prv$ after the first forward CAS of I. Similarly for $y_I.nxt = z_I$. ◀

Consider Fig. 2 and 3. By Lemma 8.3, just before the first forward CAS of I, the nxt and prv field of x_I, y_I and z_I are as shown in step 1. By Lemma 8.1, this CAS changes $x_I.nxt$ as shown in step 2 and the first backward CAS of I changes $z_I.prv$ as shown in step 3. The next lemma follows easily.

▶ **Lemma 9.** *A Node $v \neq head$ that was reachable before is reachable now iff $v.prv.nxt = v$.*

5.3 Part 3: Linearizability

Part 3 of our proof shows that operations are linearizable. The following four lemmas show that there is a linearization point for each move operation. In the following four proofs, $\langle y$, -, z, x, -, -$\rangle$ denotes the result updateCursor(c) returns on line 44 or 57. Let C_{77} be the configuration before the last execution of line 77 inside that call to updateCursor.

▶ **Lemma 10.** *If moveRight(c) changes c.node from y to z at line 60, there is a configuration during the move when $y.nxt = z$ and y is reachable.*

Proof Sketch. Invariant 1 and Lemma 9 imply that $y \neq head$ is reachable in C_{77}. (Some reasoning is required to see this, since line 77 does two reads of shared memory.) If y is reachable when $y.nxt = z$ on line 85, the claim holds. Otherwise, between C_{77} and line 85, a forward CAS of some Info object I with $I.nodes[1] = y$ made y unreachable. By Lemma 8.4, $y.nxt$ is always $I.nodes[2]$ after the CAS. Since $y.nxt = z$ on line 85, $z = I.nodes[2]$ and, by Lemma 8.3, the claim holds just before the CAS. ◀

▶ **Lemma 11.** *If moveRight(c) returns false, there is a configuration during the move when c.node.val = EOL and c.node is reachable.*

Proof Sketch. Invariant 1 and Lemma 9 imply, in C_{77}, y is reachable and the claim is true. ◀

▶ **Lemma 12.** *If moveLeft(c) changes c.node from y to v at line 49, 53 or 54, there is a configuration during the move when $v.nxt = y$ and v is reachable.*

Proof Sketch. Consider line 53. Since the move does not return on line 46, $x \neq head$. Lemma 9 implies x is unreachable after line 47. By Lemma 8.1, x became unreachable by the first forward CAS of some Info object I with $I.nodes[1] = x$. Since $x.state =$ marked on line 48, a delete created I. By Lemma 8.4, $x.nxt$ is always $I.nodes[2]$ after the forward CAS. Since $x.nxt = y$ on line 47, $y = I.nodes[2]$. Since the read of $y.prv$ returns x

on line 85, the first backward CAS of I did not occur before that read (step 2 of Fig. 2). So, at some configuration C during the move (step 2 of Fig. 2), $I.nodes[0].nxt = I.nodes[2] = y$ and $I.nodes[0]$ is reachable. Since $x.prv$ is always $I.nodes[0]$ after the forward CAS of I, the move sets w to $I.nodes[0]$ on line 51 and then sets $c.node$ to w. So, the claim holds at C. The proofs for line 49 and 54 are similar to the case above and the proof of Lemma 10, respectively. ◀

▶ **Lemma 13.** *If moveLeft(c) returns false, c.node is head.nxt in a configuration during the move.*

Proof Sketch. If moveLeft returns on line 46, the proof is similar to Lemma 10, since $x = head$ and $c.node = head.nxt$ at a configuration during the move. For line 52, the proof is similar to Lemma 12, since $w = head$ and $c.node = head.nxt$ at a configuration during the move. ◀

We now define linearization points. A move is linearized at the step after the configuration defined by Lemma 10, 11, 12 or 13. If there is a forward CAS of an Info object created by an update, the update is linearized at the first such CAS. InitializeCursor and resetCursor are linearized when they read *head.nxt*. Each get, each delete that returns false and each operation that returns invalidCursor is linearized at the first step of the last execution of line 77 inside its last call to updateCursor. Let $(\mathbb{L}, \mathbb{S})$ be an auxiliary variable of type list. When an operation is linearized, the same operation is atomically applied to $(\mathbb{L}, \mathbb{S})$ according to the sequential specification. To prove our linearization is correct, we show in Lemma 14 how the auxiliary variable $(\mathbb{L}, \mathbb{S})$ is accurately reflected in the state of the actual list, implying each operation returns the same response as the corresponding operation on $(\mathbb{L}, \mathbb{S})$.

In Lemma 14, we use abstract values to construct a one-to-one correspondence between Nodes in the list and items in $\mathbb{L}$. The *absVal* of the EOL item is 1. If an item $\mathbb{q}$ is inserted between items $\mathbb{p}$ and $\mathbb{r}$ in $\mathbb{L}$, $\mathbb{q}.absVal = (\mathbb{p}.absVal + \mathbb{r}.absVal)/2$. If $\mathbb{q}$ is inserted before the first item $\mathbb{r}$, $\mathbb{q}.absVal = \mathbb{r}.absVal/2$. The cursor in $\mathbb{S}$ corresponding to Cursor c is denoted $\mathbb{c}$. After the linearization point of a successful operation *op* called with c, *op* might update *c.node*, but $\mathbb{c}$ is updated at the linearization point. To keep track of the value of $\mathbb{c}$, we define a prophecy variable *c.updatedNode*. If a configuration C is after the linearization point of a successful operation *op* called with c but before *op* sets *c.node*, then *c.updatedNode* in C is the Node that *op* would set *c.node* to later. Otherwise, *c.updatedNode* = *c.node*. Since c is a local variable, *c.node* might become out of date when other processes update $\mathbb{c}$. The true location of a cursor c whose *c.upatedNode* is x is

$$realNode(x) = \begin{cases} realNode(x.copy) & \text{if } x.state = \text{copied and } x \text{ is unreachable,} \\ realNode(x.nxt) & \text{if } x.state = \text{marked and } x \text{ is unreachable,} \\ x & \text{otherwise.} \end{cases}$$

We say *realNodePath*(x) in configuration C is the sequence of Nodes used to define $realNode(x)$ starting with x and ending at a reachable Node. An update is *successful* if it is linearized at a forward CAS.

▶ **Lemma 14.**

1. *If operation op is linearized at step S and terminates with result r, the corresponding abstract operation applied to $(\mathbb{L}, \mathbb{S})$ atomically at S also returns r.*
2. *The sequence of abstract values and values of Nodes that are reachable (excluding head and tail) and of items in $\mathbb{L}$ are equal.*
3. *For each $\mathbb{c}$ in $\mathbb{S}$, $\mathbb{c}.item.absVal = realNode(c.updatedNode).absVal$.*

4. *$\mathbb{c}.invIns$ is true in configuration C iff (a) a Node x is on $realNodePath(c.updatedNode)$ in C such that x is copied and unreachable, or (b) C is between the invocation of an operation op called with c and op's linearization point and op's local variable $invIns$ is true.*
5. *$\mathbb{c}.invDel$ is true in configuration C iff (a) a Node x is on $realNodePath(c.updatedNode)$ in C such that x is marked and unreachable, or (b) C is between the invocation of an operation op called with c and op's linearization point and op's local variable $invDel$ is true.*

Proof Sketch. Suppose the lemma is true up to step S and C is the configuration before S. We show the lemma is true at the configuration C' after S.

Statement 1: Suppose S is the linearization point of *op* called with c.

Case 1: *op* returns invalidCursor. S is the first step of the last execution of line 77 inside *op*'s last call to updateCursor. In C, *op*'s $invDel$ or $invIns$ is true and $\mathbb{c}.invDel$ or $\mathbb{c}.invIns$ is true by induction hypothesis 4 and 5. So, the abstract operation also returns invalidCursor.

Case 2: *op* is a delete that returns false. S is the first step of the last execution of line 77 inside *op*'s last call to updateCursor. By similar argument to Case 1, $\mathbb{c}.invDel$ is false in C. We show $\mathbb{c}.item.value$ = EOL in C. In C, $c.updatedNode = c.node$. Invariant 1 and Lemma 9 imply that $c.node$ is reachable in C. In C, $\mathbb{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(c.node).absVal = c.node.absVal$ (by induction hypothesis 3). By induction hypothesis 2, $\mathbb{c}.item.value = c.node.value =$ EOL in C. So, $\mathbb{c}.item.value =$ EOL in C and the abstract deletion must also return false.

Case 3: *op* is a move, get or successful update. This case is handled similarly to Case 1 and 2.

Statement 2: By Statement 1, unsuccessful updates change neither $\mathbb{L}$ nor the reachable Nodes. By Lemma 8.1, $\mathbb{L}$ and the reachable Nodes are changed only by the first forward CAS of an Info object. Suppose S is the first forward CAS of an Info object I created by a delete(c). (A similar argument applies to insertBefore.) Since $c.updatedNode = c.node = y_I$ is reachable at C (by Lemma 8.3), $\mathbb{c}.item.absVal = y_I.absVal$ at C (by induction hypothesis 3). Only y_I becomes unreachable at C'. Likewise, only $\mathbb{c}.item$ is removed from $\mathbb{L}$ at C'.

Statement 3: Only linearization points of operations can change $realNode(c.updatedNode)$ or $\mathbb{c}.item$. Suppose S is the linearization point of *op*. We consider different cases.

Case 1: *op* is an initializeCursor or resetCursor that terminates. Then, step S is a read of $head.nxt$ on line 63 or 68. By Statement 1, S sets $\mathbb{c}.item$ to the first item in $\mathbb{L}$. Let x be the value of $head.nxt$ in C'. The $absVal$ of the first item in $\mathbb{L}$ is $x.absVal$ in C' (by Statement 2) and $\mathbb{c}.item.absVal = x.absVal$ in C'. Since *op* sets $c.node$ to x on line 63 or 68, $c.updatedNode = x$ in C'. In C', $\mathbb{c}.item.absVal = x.absVal = realNode(x).absVal = realNode(c.updatedNode).absVal$.

Case 2: *op* is called with c and returns invalidCursor or *op* is a delete(c) that returns false or *op* is a get(c) that terminates. Then, S is the first step of the last execution of line 77 inside *op*'s last call to UPDATECURSOR. By Statement 1, S does not change $\mathbb{c}.item$. By induction hypothesis 3, $\mathbb{c}.item.absVal = realNode(c.updatedNode).absVal$ in C. Since S does not change $realNode(c.updatedNode)$, the same equality holds in C'.

Case 3: *op* is a move or successful update. This case is handled similarly to Case 1.

Statement 4 and 5: We show Statement 4 is true. The proof of Statement 5 is symmetric. It is easy to show that the only steps S that we must consider are linearization points of operations, executions of line 79, 80 or 83 and invocations of an operation called with c. Suppose S is the linearization point of an operation *op* called with c. In C', $\mathbb{c}.invIns$ and Statement 4.b are false. It is easy to show that, in C', $c.updatedNode$ is reachable and it is the only Node on $realNodePath(c.updatedNode)$. So, Statement 4.a is false. The rest of cases are handled similarly. ◀

Linearizability of our implementation follows from Lemma 14.1.

6 Amortized Analysis

Our amortized analysis gives an upper bound on the total number of steps performed by all operations in any finite execution. A Cursor is *active* if it has been initialized, but not yet destroyed. Let $\dot{c}(op)$ be the maximum number of active Cursors at any configuration during operation *op*. We prove that the amortized complexity of each update *op* is $O(\dot{c}(op))$ and each move is $O(1)$. More precisely, for any finite execution α, the total number of steps in α is $O(\sum_{op \text{ is an update in } \alpha} \dot{c}(op) + \sum_{op \text{ is a move in } \alpha} 1)$. It follows that the implementation is non-blocking. The analysis (about 20 pages long in [19]) is quite complex, so we only provide the intuition here. Parts of it are similar to the analysis of search trees in [6] but the parts dealing with Cursors and moves are original. We simplified the analysis using the potential method and show how to generalize the analysis of [6] to handle operations that flag more than two nodes.

We first bound the number of iterations of line 77–83. Each update *op* deletes or replaces at most one Node. Any Cursor c whose true location (as defined in Sec. 5) is at that Node when *op* is performed will have to perform one iteration of line 77–83 when updateCursor(c) is next called to follow the *nxt* or *copy* pointer of the Node. Since there are at most $\dot{c}(op)$ such cursors, the total number of iterations of line 77–83 in the execution is at most $\sum_{op \text{ is an update}} \dot{c}(op)$.

Each iteration of line 18–30 or 32–42 inside an update is called an *attempt*, which is *successful* if it returns on line 20, 30, 34 or 42, or *unsuccessful* otherwise. Excluding calls to updateCursor (which have already been accounted for), each attempt of an update takes $O(1)$ steps and, each move and each get operation take a total of $O(1)$ steps. It follows that the amortized complexity of a move and a get is $O(1)$. It remains to prove that the total number of unsuccessful attempts in the execution is $O(\sum_{op \text{ is an update}} \dot{c}(op))$. An attempt is unsuccessful because one of the Nodes to be flagged is either (1) observed to be marked or copied when checkInfo returns false on line 92 or (2) flagged by another update.

First, consider attempts that fail for reason (1). Consider a Cursor c that is active when an update *op* sets $x.state$ to copied or marked. This causes at most two attempts of c's updates to fail: if an attempt of an update op' fails when reading $x.state$ at line 92, line 89 of the next attempt ensures *op* is completed and no subsequent attempt reaches x. To pay for these attempts, *op* stores $2\dot{c}(op)$ of potential when *op* sets $x.state$. There is one other possibility: an operation op' might be called with a Cursor that is created *after* *op* set $x.state$ to marked or copied. Again, at most two attempts of op' might fail because of reading $x.state$ at line 92. To pay for these attempts, op' stores $2\dot{c}(op')$ of potential when it is invoked, since there are at most $\dot{c}(op')$ reachable Nodes that are marked or copied when op' begins. Thus, the total number of attempts that fail for reason (1) is $O(\sum_{op \text{ is an update}} \dot{c}(op))$.

Bounding the number of attempts that fail for reason (2) is the most intricate part of the analysis. An attempt *att* of an update may fail because a Node it wishes to flag gets flagged by an attempt att' of another operation, causing *att*'s test at line 88 or 94 to fail or *att*'s flag CAS on line 99 to fail. If att' were guaranteed to succeed in this case, the analysis would be simple. However, att' itself may also fail because it is blocked by the attempt of a third operation, and so on. Since a successful flag CAS might belong to an unsuccessful attempt, such a step does not store any potential. However, $O(\dot{c}(op))$ of potential is stored

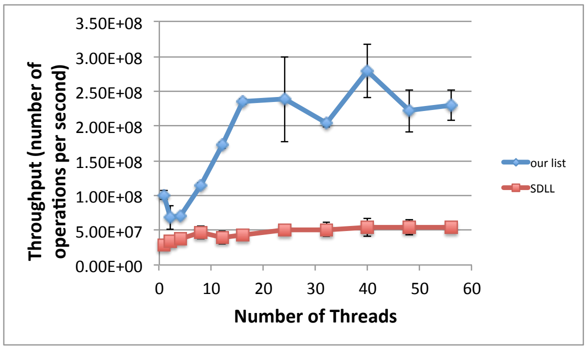


Figure 6 Comparison of our list and the list in [21].

(a) when an update *op* is invoked, (b) when a forward or backward CAS step of *op* succeeds and (c) when the *status* of an Info object created by *op* is set to committed. We prove that this potential is sufficient to pay for any attempt that fails for reason (2). Thus, the total number of attempts that fail for reason (2) is also $O(\sum_{op \text{ is an update}} \dot{c}(op))$. It follows the amortized complexity of an update *op* is $O(\dot{c}(op))$.

7 Concluding Remarks

A correctness proof is essential for data structures such as ours since it is not possible to test all possible executions. Writing detailed correctness proofs helped us to correct bugs in earlier versions and then verify the correctness of our list. It also helped us to simplify the pseudo-code and improve its complexity.

Our amortized bound of $O(\dot{c}(op))$ for an update *op* is quite pessimistic: the worst case happens only if concurrent updates are scheduled in a very particular way. We expect our list would have even better performance in practice. Our experimental results suggest that on an Intel Xeon multi-core machine, a Java implementation of our list scales well and also outperforms the list in [21] (SDLL). We used our own Java implementations for both our list and SDLL. Each data point in Figure 6 is the average of eight 4-second trials in which each thread continuously performs on-average 100 move operations and then one update operation on a list with 200 items. Our results show that the throughput of SDLL is not improved when the number of threads is increased, which agrees with the empirical results presented in [21].

In our approach, as in [3, 4, 5, 8, 18], updates create Info objects and duplicate Nodes, which induces some overhead. Despite such overheads, empirical evaluations in [4, 5, 18] and here confirm the practicality and scalability of this technique.

Though moves have constant amortized time, they are not wait-free. For example, if cursors c and c' point to the same node, a moveLeft(c) may never terminate if an infinite sequence of insertions at c' succeed, since the updateCursor of the move could run forever.

Future work includes designing shared cursors. Generalizing our coordination scheme could provide a simpler way to design non-blocking data structures. Although the proof of correctness and analysis is complex, it is modular, so it could be applied more generally. Our help routine gives a general way of coordinating operations that make several changes to a data structure. Parts 1 and 2 of the proof are primarily about this routine and could be reused for other data structures. Detailed arguments about linearizability of the operations (Part 3 of the proof) would likely depend more on the data structure being implemented.

Acknowledgements. I thank my supervisor, Eric Ruppert, for his great guidance and advice, and the management, staff, and facilities of the Intel Manycore Testing Lab.[1]

References

1 Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.

2 Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, SPAA'93, pages 261–270, 1993.

3 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, PODC'13, pages 13–22, 2013.

4 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming*, PPOPP'14, pages 329–342, 2014.

5 Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proceedings International Conference on Principles of Distributed Systems*, OPODIS'11, pages 207–221, 2011.

6 Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, PODC'14, pages 332–340, 2014.

7 Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, PODC'12, pages 115–124, 2012.

8 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC'10, pages 131–140, 2010.

9 Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, July 1990.

10 Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, PODC'02, pages 260–269, 2002.

11 Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition.* 2010.

12 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC'02, pages 265–279, 2002.

13 Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000. See `http://babelfish.arc.nasa.gov/trac/jpf`.

14 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[1] `http://www.intel.com/software/manycoretestinglab`
`http://www.intel.com/software`.

15 Jikuan Hu and Weiqing Wang. Algorithm research for vector-linked list sparse matrix multiplication. In *Proceedings of the 2010 Asia-Pacific Conference on Wearable Computing Systems*, APWCS'10, pages 118–121, 2010.

16 Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, SPAA'03, pages 314–323, 2003.

17 Matthias Pfeffer, Theo Ungerer, Stephan Fuhrmann, Jochen Kreuzinger, and Uwe Brinkschulte. Real-time garbage collection for a multithreaded Java microcontroller. *Real-Time Systems*, 26(1):89–106, January 2004.

18 Niloufar Shafiei. Non-blocking Patricia tries with replace operations. In *Proceedings of the 33rd International Conference on Distributed Computing Systems*, ICDCS'13, pages 216–225, 2013.

19 Niloufar Shafiei. *Non-blocking data structures handling multiple changes atomically.* PhD thesis, Department of Electrical Engineering and Computer Science, York University, Toronto, Canada, August 2015.

20 Håkan Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming*, 39(6):694–716, 2011.

21 Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

22 John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, PODS'92, pages 212–222, 1992.

23 John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 214–222, 1995.

Poly-Logarithmic Adaptive Algorithms Require Unconditional Primitives

Hagit Attiya[*1] **and Arie Fouren**[†2]

1 **Department of Computer Science, Technion, Haifa 32000, Israel**
`hagit@cs.technion.ac.il`

2 **Faculty of Business Administration, Ono Academic College, Kiryat Ono, 5545173, Israel**
`aporan@ono.ac.il`

Abstract

This paper studies the step complexity of adaptive algorithms using primitives stronger than reads and writes. We first consider *unconditional* primitives, like fetch&inc, which modify the value of the register to which they are applied, regardless of its current value. Unconditional primitives admit snapshot algorithms with $O(\log k)$ step complexity, where k is the total or the point contention. These algorithms combine a renaming algorithm with a mechanism for propagating values so they can be quickly collected.

When only *conditional* primitives, e.g., compare&swap or LL/SC, are used (in addition to reads and writes), we show that any collect algorithm must perform $\Omega(k)$ steps, in an execution with total contention $k \in O(\log \log n)$. The lower bound applies for snapshot and renaming, both one-shot and long-lived. Note that there are snapshot algorithms whose step complexity is polylogarithmic in n using only reads and writes, but there are no adaptive algorithms whose step complexity is polylogarithmic in the contention, even when compare&swap and LL/SC are used.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.4.1 Process Management

Keywords and phrases collect, atomic snapshot, renaming, fetch&inc, compare&swap

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.36

1 Introduction

Collecting up-to-date information from all processes is a key to coordination and synchronization, for example, for implementing *atomic snapshots* [1] or solving *renaming* [7]. A simple way to do so is to have an array where each process *stores* its latest value in the entry associated with its index, and to read this array in order to *collect* the values of all processes.

This scheme is an overkill when only a few processes participate in the algorithm: many entries are read from the array although they contain irrelevant information about processes not wishing to coordinate. Better performance is achieved when the step complexity depends only on the *total contention*, namely, the number of processes that participate in the algorithm. We say that such an algorithm is *adaptive to total contention*. Even better is an algorithm whose step complexity is adaptive to *point contention*, which is the maximal number of processes simultaneously executing the algorithm concurrently.

[*] This work is supported by the Israel Science Foundation (grant number 1749/14), and by Yad HaNadiv foundation.

[†] This work is supported by the Ono Academic College Research Fund.

19th International Conference on Principles of Distributed Systems (OPODIS 2015).
Editors: Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Gradinariu; Article No. 36; pp. 36:1–36:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Algorithm	Problem	Step Complexity	Contention	Primitives
Algorithm 1, using [22]	atomic snapshot	$O(\log(k))$	point	LL/SC, fetch&inc, bounded-fetch&dec
Algorithm 1, using fetch&inc	atomic snapshot	$O(\log(k))$	total	LL/SC, fetch&inc
Algorithm 2	atomic snapshot	$O(\min(k, \log n))$	point	LL/SC

Figure 1 Summary of upper bounds for long-lived atomic snapshots.

There has been significant progress in designing adaptive collect algorithms that only use reads and writes. These algorithms have step complexity that is at least linear in the total or point contention [3, 12, 2]. They have been used in algorithms for atomic and immediate snapshots, renaming and timestamping [9, 4, 10, 12, 11].

Much less is known about the complexity of adaptive collect using primitives stronger than reads and writes. A notable exception is the collect algorithm of Moir et al. [17], which uses test&set. This algorithm uses less memory than collect algorithms that use only reads and writes, but its step complexity is not much better, as it is linear in the point contention.

This paper studies the step complexity of adaptive algorithms using primitives stronger than reads and writes and investigates whether they can be used to obtain adaptive algorithms with poly-logarithmic step complexity.

We present a snapshot algorithm with $O(\log k)$ step complexity, where k is the total contention; the algorithm uses fetch&inc (as well as LL/SC). We also describe a snapshot algorithm with $O(\log k)$ step complexity, where k is point contention, using fetch&inc and bounded-fetch&dec primitives (and LL/SC). These algorithms combine a renaming algorithm for having processes obtain unique locations to store their values, with a mechanism for propagating these values up a tree ("bubbling" them up), so they can be quickly collected. (See Table 1.)

These algorithms use *unconditional* primitives, like fetch&inc and bounded-fetch&dec, which modify the value of the register to which they are applied regardless of the current value of the register. In contrast, *conditional* primitives [15], like compare&swap and LL/SC, modify the register only if it holds a specific value that depends on the input of the conditional operation. When only conditional primitives are used (in addition to reads and writes), we show that any collect algorithm must perform $\Omega(k)$ steps, in an execution with total contention $k \in O(\log \log n)$. Specifically, the bound applies for the sum of the step complexities of a pair of store and collect operations performed by some process on a one-shot collect object.

There is a trade-off between the contention k and the step complexity lower bound of collect. If the total contention k is in $\Theta(n)$, we prove that at least $O(\log k)$ steps are required to perform store and collect. We present an algorithm with $O(\min(k, \log n))$ step complexity, matching the lower bounds.

Clearly, the lower bound for collect immediately applies for snapshots. Moreover, the reduction from collect to renaming, described earlier, implies that the lower bound also applies to renaming, giving an alternative proof of the $\Omega(k)$ step complexity lower bound previously proved by Alistarh et al. [5]. Obviously, it also implies a $\Omega(k)$ lower bound for the long-lived versions of collect, snapshot and renaming, where k is point contention.

Related Work. The work of Fich, Hendler and Shavit [15] showed a separation (in terms of space complexity) between conditional and unconditional primitives. It proves that any wait-free implementation of a large class of objects that includes counters, stacks, queues and snapshots from reads, writes and conditional primitives requires at least $\Omega(n)$ space. The same space is required for any starvation-free mutual exclusion implemented from conditional primitives. In contrast, using unconditional primitives (like fetch&add) allows to implement these objects and to solve mutual exclusion using only constant space. These results align with the separation in the terms of time complexity presented in our work.

There is an $\Omega(k)$ lower bound on the step complexity of adaptive *mutual exclusion* using reads, writes and compare&swap [21].[1] The lower bound requires that the total number of the processes $n = \Omega(k^{2^k})$, which is slightly higher than the requirement on n in our lower bound.

Alistarh et al. [5] show how to transform any adaptive wait-free renaming with sub-exponential name space $M(k)$ into adaptive mutual exclusion with $O(\log(M(k)) = o(k)$ additional steps. Using the $\Omega(k)$ lower bound on the step complexity of adaptive mutual exclusion [21], they derive an $\Omega(k)$ lower bound on the step complexity of adaptive renaming. This lower bound also requires $n = \Omega(k^{2^k})$.

Another reduction from mutual exclusion to renaming can be obtained using unbalanced tournament tree presented in [8]. For sub-exponential name space $M(k)$ this transformation also requires $O(\log(M(k)) = o(k)$ additional steps. Combined with the $\Omega(k)$ lower bound for mutual exclusion [21], it also provides an $\Omega(k)$ step complexity lower bound for adaptive renaming.

There is a simple implementation of $O(k^2)$-renaming from lattice agreement, with $O(1)$ additional steps [16, Algorithm 11]. Together with the $\Omega(k)$ step complexity lower bound for adaptive renaming [5, 8] it implies an $\Omega(k)$ step complexity lower bound for adaptive lattice agreement (and therefore, for atomic and immediate snapshots).

We could not find a reduction with sublinear step complexity from these problems (mutual exclusion, renaming or snapshots) to adaptive store / collect, even with compare&swap and LL/SC. Without a way to deduce the lower bound for adaptive store-collect from the existing lower bounds for mutual exclusion or renaming, we had to directly prove the lower bound for adaptive collect presented in this paper.

2 The Computation Model

In the *wait-free asynchronous shared-memory* model, n *processes*, $p_0, \ldots, p_{n-1}$ communicate by applying *primitive operations* (in short, *primitives*) to shared memory registers [21]. A process is described as a state machine, with a set of (possibility infinite number of) *states*, one of which is a designated *initial state*, and a state transition function.

The executions of the system are sequences of events. In each *event*, based on its current state, a process applies a primitive to a shared memory register and then changes its state, according to the state transition function. At the beginning of the execution, all shared registers hold the value $\perp$. During an execution, no process ever changes the value of a shared register to $\perp$.

An event ϕ in which a process p applies a primitive op to register R is denoted by a triple $\langle p, R, op \rangle$. An *execution* α is a (finite or infinite) sequence of events $\phi_0, \phi_1, \phi_2, \ldots$. There are no constraints on the interleaving of events by different processes, reflecting the assumption

[1] This lower bound holds also for LL/SC by using an implementation of LL/SC from compare&swap, with constant step complexity [19].

that processes are asynchronous. We denote by *next_event*(p_i, α) the next event a process p_i will perform if it is scheduled to take a step after an execution prefix α.

For an execution α and a set of processes P, $\alpha|_P$ is the sequence of all events in α by processes in P; $\alpha|_{\overline{P}}$ is the sequence of all events in α that are *not* by processes in P. If $P = \{p\}$, we write $\alpha|_p$ instead of $\alpha|_{\{p\}}$ and $\alpha|_{\overline{p}}$ instead of $\alpha|_{\overline{\{p\}}}$. An execution α is P*-only* if $\alpha = \alpha|_P$, and it is P*-free* execution if $\alpha = \alpha|_{\overline{P}}$. Two executions α and α' are *equivalent* with respect to P if $\alpha|_P = \alpha'|_P$.

We always assume the availability of read and write primitives. A read(R) primitive returns the current value of R and does not change its value. A write$(v,\ R)$ operation sets the value of R to v, and does not return a value. We make no restrictions on which process can read from or write to each register, i.e., they are *multi-writer multi-reader.*

A fetch&inc on register R atomically increments the value of the register by 1 and returns the previous value. That is, if the value of R immediately before the invocation of fetch&inc (R) was v, then the primitive sets the value of R to $v+1$ and returns v. Similarly, fetch&dec atomically decrements the value of the register by 1 and return the previous value. A bounded-fetch&dec primitive is similar to standard fetch&dec, except that if the value of register R is 0 before the primitive is applied to it, then the value of R remains unchanged.

LL on register R returns the current state of R. SC $(R,\ v)$ invoked by a process p changes the state of R to v only if no other process has changed the value of R since the the latest execution of LL (R) by p. If the value of R is changed SC returns **true**, otherwise it returns **false**.

A compare&swap$(R,\ v,\ u)$ primitive works as follows. If the register R holds the value v, then the state of R is changed to u and and **true** is returned (the compare&swap *succeeds*). Otherwise, the state of R remains unchanged and **false** is returned (the compare&swap *fails*).

An *implementation* of a high-level object provides algorithms for each high-level operation supported by the object. Some of the transitions are *requests*, invoking a high-level operation, or *responses* to a high-level operation. When a high-level operation is invoked, the process executes the algorithm associated with the operation, applying primitives to the shared registers, until a response is returned.

In a *well-formed* execution, a high-level operation is invoked only if there is a response to the previous high-level operation, that is, a process alternates between invocations and *matching* responses, beginning with an invocation. A well-formed execution α defines a partial order on operations: If the response of operation op_1 occurs in α before the invocation of operation op_2, then op_1 *precedes* op_2 and op_2 *follows* op_1. We say that op_1 and op_2 are *non-overlapping.*

We require implementations to be *linearizable* [18]. Roughly speaking, a linearizable object guarantees that there is a reordering of the object operations which satisfies the sequential specification of the object and respects the real-time order of non-overlapping operations among all the processes.

Adaptive Algorithms. Let α' be a finite prefix of an execution α. Process p_i performing a high-level operation *op* is *active* at the end of α', if α' includes an invocation of *op* without a return from *op*. The set of the processes active at the end of α' is denoted *active*(α'). The *point contention* at the end of α', denoted *pointCont*(α'), is $|active(\alpha')|$.

The *total contention* during α is the total number of processes active in α:

$$totalCont(\alpha) = \left| \bigcup_{\alpha' \text{ prefix of } \alpha} active(\alpha') \right| .$$

Assume that β is a finite interval β of α, i.e., $\alpha = \alpha_1\beta\alpha_2$. The point contention during β, denoted *pointCont*(β), is the maximum contention in all prefixes $\alpha_1\beta'$ of $\alpha_1\beta$:

$$\mathit{pointCont}(\beta) = \max_{\alpha_1\beta' \text{ prefix of } \alpha_1\beta} \mathit{pointCont}(\alpha_1\beta') \ .$$

Consider an execution α of an algorithm A implementing a high-level operation *op*. For process p_i executing operation op_i, $\mathit{step}(A, \alpha, op_i)$ is the number of operations on shared registers p_i performs executing op_i in α. The step complexity of A in α, denoted $\mathit{step}(A, \alpha)$, is the maximum of $\mathit{step}(A, \alpha, op_i)$ over all operations op_i of all processes p_i.

Consider a bounded function $\mathrm{S} : \mathcal{N} \mapsto \mathcal{N}$. An algorithm implementing operation *op* is S-*adaptive to total contention* if for every execution α and every operation op_i with interval β_i, $\mathit{step}(A, \alpha) \leq \mathrm{S}(\mathit{totalCont}(\alpha))$. That is, the step complexity of the algorithm in any execution is bounded by a function of the total contention during the execution. An algorithm implementing operation *op* is S-*adaptive to point contention* if for every execution α and every operation op_i with interval β_i, $\mathit{step}(A, \alpha, op_i) \leq \mathrm{S}(\mathit{pointCont}(\beta_i))$. That is, the step complexity of an operation op_i with interval β_i is bounded by a function of the point contention during β_i.

Since contention is bounded by n, an operation op_i of p_i terminates within a bounded number of steps of p_i, regardless of the behavior of other processes; that is, adaptive algorithms are *wait-free*.

3 Problems Studied in this Paper

Collect. A solution for the *collect* problem should define algorithms for two operations – store and collect. Intuitively, a store(*val*) operation of p_i declares *val* as the latest value for p_i, and a collect operation returns a *view* containing the latest values stored by active processes. A *view* is a set of process-value pairs, $V = \{\langle p_{i_1}, v_{i_1}\rangle, \ldots\}$, without repetitions of processes. $V(p_j)$ refers to v_j, if $\langle p_j, v_j\rangle \in V$, and to $\perp$ otherwise.

A collect operation *cop* returns a view V such that the following holds for every process p_j:

Validity: If $V(p_j) = \perp$, then no store operation of p_j precedes *cop*; if $V(p_j) = v \neq \perp$ then v is the value of a store operation *sop* of p_j that does not follow *cop*, and there is no other store operation *sop*$'$ of p_j that follows *sop* and precedes *cop*.

That is, *cop* does not read from the future or miss a preceding store operation.

Moreover, if a collect operation *op* follows another collect operation *cop*$'$, then *cop* should return a view which is more up-to-date. To capture this notion, we define a partial order on views: $V_1 \preceq V_2$, if for every process p_i such that $\langle p_i, v_i^1\rangle \in V_1$, we have $\langle p_i, v_i^2\rangle \in V_2$, and v_i^2 is written in a store operation of p_i that follows or is equal to a store operation of p_i which writes v_i^1. Using this definition, we formulate the property of the collect problem as follows:

Regularity: Assume a collect operation *cop* by p_i returns V_1, and a collect operation *cop*$'$ by p_j returns V_2. If *cop* precedes *cop*$'$, then $V_1 \preceq V_2$.

Atomic Snapshots. The *atomic snapshot* problem [1] extends the collect problem by requiring views to look instantaneous. We assume a combined upscan operation, which updates a new value and atomically collects a view. The returned views should satisfy the following conditions:

Validity: If an upscan operation op returns a view V, and precedes an upscan operation op', then V does not include the value written by op'.[2]

Self-inclusion: The view returned by the ℓth upscan operation of p_j includes the ℓth value written by p_j.

Comparability: If V_1 and V_2 are the views returned by two upscan operations, then either $V_1 \preceq V_2$ or $V_2 \preceq V_1$.

The *lattice agreement* problem is a special case of atomic snapshots, in which a process performs the algorithm at most once, writing its own identifier and collecting a view which contains identifiers of participating processes. The returned views should satisfy the validity, self-inclusion and comparability properties of the atomic snapshot.

Immediate Snapshots. The *immediate snapshot* problem [14] is an extension of the atomic snapshot problem; it supports a combined im-upscan operation, which updates a new value and returns a view. In addition to the validity, self-inclusion, and comparability properties of the atomic snapshot problem, returned views should satisfy the next condition:

Immediacy: If the view returned by some im-upscan operation, V_1, includes the value written in the ℓth im-upscan of p_j which returns the view V_2, then $V_2 \preceq V_1$.

M-Renaming. In the *long-lived M-renaming* problem, processes $p_1, \ldots, p_n$ with unique names from the range $\{0, \ldots, N-1\}$ repeatedly acquire and release distinct names in the range $\{0, \ldots, M-1\}$. The range $\{0 \ldots N-1\}$ is the *initial name space*, and the range $\{0 \ldots M-1\}$ is the *final name space*. A solution supplies two procedures: getName returning a *new name*, and releaseName; p_i alternates between invoking getName$_i$ and releaseName$_i$, starting with getName$_i$.

For the long-lived renaming problem we redefine the notion of an active process. Process p_i is *active* at the end of execution prefix α', if α' includes an invocation of getName$_i$ without a return from the matching releaseName$_i$. A long-lived renaming algorithm should guarantee *uniqueness* of new names: Active processes hold distinct names at the end of α'.

A renaming algorithm has a name space *adaptive to point contention*, if there is a function M, such that the name obtained in an interval β of getName is in the range $\{1, \ldots, \mathrm{M}(\mathit{pointCont}(\beta))\}$. The name space is adaptive to *total* contention, if the new names are in the range $\{1, \ldots, \mathrm{M}(\mathit{totalCont}(\beta))\}$.

One-shot M-renaming is a special case of long-lived renaming. The processes start with unique names from the range $\{0, \ldots, N-1\}$ and are required to choose distinct names in the range $\{0 \ldots M-1\}$, where $M < N$.

4 Sub-linear Adaptive Algorithms for Atomic Snapshots

4.1 Atomic Snapshots Using Renaming

This section presents a modular construction of atomic snapshots using renaming. The algorithm uses an unbalanced binary tree, consisting of a sequence of complete binary trees of growing sizes, connected as shown in Figure 2. (This tree structure was also used in [6, 22].)

In Algorithm 1, a process starts by acquiring a name i using an adaptive renaming algorithm (Line 1.2). Then it exclusively accesses the i-th leaf of the unbalanced binary tree,

[2] Typically, this condition trivially holds and we do not prove it below.

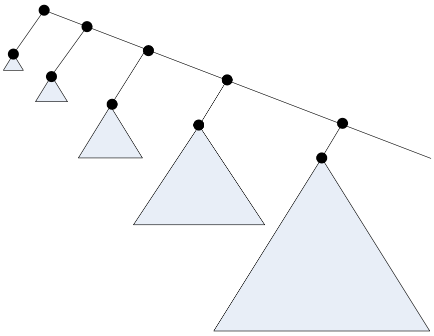


Figure 2 Unbalanced binary tree used in Algorithm 1.

writes its information in the register associated with the leaf. Then, it climbs up to the root, updating the views associated with the nodes on the path. (This part of the algorithm is similar to the f-array implementation of Jayanti [20].) After updating the root, the process releases the acquired name.

The next lemma shows that the sequence of the views stored in a node is monotonically increasing:

▶ **Lemma 1** (Comparability). *Let $V_1, V_2, \ldots$ be the sequence of the views written into v.subtree-View of some node v in Line 1.3, 1.17 or 1.17. Then $V_1 \preceq V_2 \preceq \ldots$.*

Proof. If node v is a leaf, then define a sequence of execution intervals $\gamma_1, \gamma_2, \ldots$ as follows: γ_i starts in the ith time a process p acquires the name associated with leaf v (Line 1.2), and it ends when p releases this name (Line 1.6). By the uniqueness of renaming, the intervals $\gamma_1, \gamma_2, \ldots$ do not overlap.

If v is an inner node, let $\gamma_1, \gamma_2, \ldots$ be the execution intervals corresponding to the pairs of successful LL/SC primitives that write views $V_1, V_2, \ldots$ into *v.subtree-View*. That is, for each successful pair SC_i, the interval γ_i starts with the corresponding LL (Line 1.16 or 1.18) and ends with the corresponding SC (Line 1.17 or 1.19). Clearly, the intervals $\gamma_1, \gamma_2, \ldots$ do not overlap.

In both cases, processes access node v in non-overlapping intervals $\gamma_1, \gamma_2, \ldots$. Therefore, a view V_i written by a process p into *v.subtree-View* in interval γ_i, (in Line 1.3 if v is a leaf, or in Line 1.17 or 1.19 if v is an inner node) is read after that by a process q in the next interval γ_{i+1} (in Line 1.16 or 1.18). Therefore, it is included in the view V_{i+1} written by q into *v.subtree-View* (in Line 1.3, Line 1.17 or 1.19). This implies that $V_i \preceq V_{i+1}$, and the lemma follows by induction on i. ◀

The following lemma states that the views returned by scan operations satisfy the self-inclusion property of atomic snapshot.

▶ **Lemma 2** (Self-inclusion). *Assume p_i performs* update(id_i, val_i)*, and then* scan() *that returns view V_i. Then $val_i \in V_i$.*

Proof. We show by induction on the sequence of the nodes visited by p_i during refresh that $val_i \in$ *v.subtree-View* after p_i leaves node v (Line 1.21).

If node v is a leaf, then p_i appends val_i to *v.subtree-View* in Line 1.3. Therefore, $val_i \in$ *v.subtree-View* after p_i leaves v.

Algorithm 1 Adaptive atomic snapshot algorithm using renaming

```
Type:
  node :
    subtree-View: view of the values stored in the subtree of the node, initially $\emptyset$
    left-child: pointer to left child node
    right-child: pointer to right child node
    parent: pointer to the parent node
Global variables:
  $T$: unbalanced binary tree of nodes
Local variables:
  $id$ : process id
  $val$ : value to be written; for simplicity we assume increasing numbers

procedure update($id$, $val$)
  $i$ = acquire-name()          ▷ using any adaptive long-lived renaming algorithm
  $v$ = $i$-th leaf of the unbalanced tree $T$ ▷ start at the bottom of the unbalanced tree
  $v.subtree\text{-}View$ = merge($v.subtree\text{-}View$, $\{\langle id, val\rangle\}$)   ▷ update your value in the leaf
  refresh($v$)          ▷ start from the current leaf $v$ and ascend back to the root
                        ▷ updating the views in the nodes along the path
  release-name($i$)          ▷ release the name associated with the $i$-th leaf
end procedure

procedure scan
  return($root.subtree\text{-}View$)          ▷ return the view from the root node
end procedure

procedure merge(views $V_1$, $V_2$, $\ldots$)   ▷ return the view of the latest processors' values
  return($\{\langle p_i, v_i\rangle \mid v_i = \max(V_1(p_i), V_2(p_i), \ldots)\}$)
end procedure

procedure refresh(node $v$)          ▷ start at leaf $v$ and ascend back to the root
  while $v \neq root$ do          ▷ updating the views in the nodes along the path
    $view$ = LL($v.subtree\text{-}View$)
▷ merge the views stored $v$ and in both its children and try to store it in $v.subtree\text{-}View$
    if $\neg$ SC($v.subtree\text{-}View$, merge($view$, $v.right\text{-}child.View$,$v.left\text{-}child.View$) then
      $view$ = LL($v.subtree\text{-}View$)          ▷ if the previous store failed, try once more
      SC($v.subtree\text{-}View$, merge($view$, $v.right\text{-}child.View$,$v.left\text{-}child.View$))
    end if
    $v$ = $v.parent$          ▷ climb up to the patent node even if both updates failed
  end while
end procedure
```

For the induction step, assume that $val_i \in$ *subtree-View* in the left child or the right child of the current node v. Process p_i reads these views before it attempts to write the merged view into *v.subtree-View* with SC (Line 1.17 or 1.19). If one of the SC primitives succeeds, then $val_i \in$ *v.subtree-View* after p_i leaves node v.

If both of these SC primitives fail, then there is a successful pair LL_j and SC_j by some process p_j such that LL_j starts after the first LL of p_i and SC_j ends before the second SC of p_i. Process p_j reads *v.left-child.subtree-View* and *v.right-child.subtree-View* (one of them containing val_i) between LL_j and SC_j. Therefore, SC_j writes a view containing val_i into *v.subtree-View*. By Lemma 1, *v.subtree-View* is monotonically increasing. Therefore, $val_i \in$ *v.subtree-View* after p_i leaves node v.

Process p_i completes refresh after leaving the root. Therefore, $val_i \in$ *root.subtree-View* after p_i returns from update. By Lemma 1, *root.subtree-View* is monotonically increasing. Therefore, the view V_i returned by the following scan operation contains val_i. ◀

By Lemmas 1 and 2 we have that Algorithm 1 implements a long-lived atomic snapshot. Let $M(k)$ be the size of the name space of the adaptive renaming algorithm used in the algorithm. The distance from the $M(k)$-th leaf of the unbalanced tree to the root is $O(\log M(k))$. In each node on the path from the leaf to the root in update, a process performs a constant number of steps. Therefore, the step complexity of the resulting snapshot algorithm is $f(k) + O(\log M(k))$, where $f(k)$ is the step complexity of the adaptive renaming algorithm.

A simple way to do renaming in Line 1 is by applying fetch&inc to a shared register. When a process executes update for the first time, it performs fetch&inc to get a name, and then it uses this name in all the following update operations. The names obtained in this way are in $0, \ldots, k-1$, where k is the total contention. Then the depth of the leaf acquired is $O(\log k)$, and therefore the step complexity of the algorithm is $O(\log k)$, where k is the total contention. The algorithm uses only LL/SC and fetch&inc.

Alternatively, we can use the long-lived adaptive k-renaming of Moir and Anderson [22, Fig. 8]. The step complexity of this algorithm is $O(\log k)$, where k is the point contention. However, it uses bounded-fetch&dec in addition to the more standard LL/SC and fetch&inc. Using this algorithm gives a $O(\log(k))$ long-lived atomic snapshot, where k is point contention, using LL/SC, fetch&inc and bounded-fetch&dec.

4.2 Atomic Snapshot with Conditional Primitives

What is the best step complexity we can achieve with only conditional primitives? When the number of participants is high ($k \sim n$), at least $\Omega(\log n)$ steps are required to perform collect (and therefore atomic snapshot) [13]. This section presents a snapshot algorithm with $O(\min(k, \log n))$ step complexity using only reads, writes and LL/SC. In the next section we prove that at least $\Omega(k)$ steps are required when contention is low ($k \in O(\log\log n)$), and at least $\Omega(\log k)$ steps is required when contention is high ($k \in \Theta(n)$). The algorithm matches the both lower bounds.

Algorithm 2 is a modification of Algorithm 1. It uses an unbalanced tree with the same structure, but the first $\log n$ leaves of the tree are reserved for processes that obtain new names in $\log n$*-restricted* adaptive k-renaming. This restricted renaming algorithm guarantees that if the total (or point) contention is less than $\log n$, then all the processes get new names in range $0, \ldots, k-1$. If the contention is higher than $\log n$, then a process p_i either gets a name in range $0, \ldots, \log n$ or a special **fail** value. If p_i gets **fail** then it accesses the unbalanced tree using its original name id_i, starting at leaf $\log n + id_i$.

Algorithm 2 Adaptive atomic snapshot with $O(\min(k, \log n))$ step complexity, using reads, writes and LL/SC.

```
procedure update($id$, $val$)
    $i$ = acquire-name()        ▷ using long-lived $k$-renaming restricted to $\log n$ names
    if $i$ == failed then
        $i = \log n + id_i$     ▷ if failed to get a name $\leq \log n$, use its original name $+ \log n$
    end if
    $v$ = $i$-th leaf of the unbalanced tree $T$     ▷ start at the bottom of the unbalanced tree
    $v.subtree\text{-}View$ = merge($v.subtree\text{-}View$, $\{val\}$)     ▷ update your value in leaf's subtree view
    refresh($v$)                ▷ start from the current leaf $v$ and ascend back to the root
                                ▷ updating the views in the nodes along the path
    if $i < \log n$ then
        release-name($i$)       ▷ release the name if it was acquired by the adaptive renaming
    end if
end procedure
```

The correctness of the algorithm follows by the uniqueness of the new names and by Lemmas 1 and 2. We can get a restricted renaming adaptive to point contention by using a sequence of $\log n$ LL/SC variables. A process sequentially accesses these variables until it succeeds to acquire one of them. If the process fails in all $\log n$ variable, it returns **failed**. This is essentially the k-renaming algorithm using test&set [22, Theorem 4], restricted to the first $\log n$ names. The step complexity of the renaming is $O(k)$, adaptive to point contention. Therefore, if $k < \log n$, each process gets a name $\leq k$ in $O(k)$ steps, and accesses a leaf at depth $O(\log k)$. Therefore the total number of steps is $O(k)$. If $k \geq \log n$, then a process accesses the tree using its original name at a leaf on depth $O(\log n)$. Thus the total complexity is $O(\log n)$. This implies that Algorithm 2 correctly implements atomic snapshot with $O(\min(k, \log n))$ step complexity, where k is the point contention.

5 Lower Bounds on Adaptive Collect with Conditional Primitives

This section proves a trade-off between the total contention and the step complexity of adaptive one-time collect using conditional primitives. For low contention, $k \in O(\log \log n)$, at least $\Omega(k)$ steps are required to complete an update operation followed by a collect. If the contention is high, $k \in \Theta(n)$, then at least $\Omega(\log k)$ steps are required to complete this pair of operations. Algorithm 2 presented in Section 4.2 solves the atomic snapshot problem using only conditional primitives (writes, reads and LL/SC) with $O(\min(k, \log n))$ step complexity, thus matching both lower bounds.

To prove the lower bounds, we construct an execution in which each active process performs a store followed by a collect, and show that at least one process performs the required number of steps. In the execution, the active processes are divided to *visible* and *invisible*. Intuitively, an *invisible* process may be removed from the execution without affecting the steps of the others. Formally, process p is *invisible* after an execution prefix α if α and $\alpha|_{\overline{p}}$ are equivalent with respect to any process in $\text{active}(\alpha) - \{p\}$. The set of the processes invisible after α is denoted $invisible(\alpha)$.

The construction proceeds in rounds. In each round, every process that is still invisible after the previous round performs its next computational event. After constructing the new round, we keep in the execution all the invisible processes, and some of the processes that became visible; the rest are deleted retroactively.

Provided that the initial number of processes is sufficiently high (as stated below), we inductively build r rounds of execution so that at least two processes p and q remain invisible after the last round. By the validity property of collect, it is impossible that two processes complete their store and collect operations without being aware of each other. This implies that at least one of them does not complete its collect in r rounds, implying that it takes at least r steps.

Choosing the number of processes which are allowed to become visible in each round, we can trade the level of contention, k, and the number of rounds, r. If we allow at most one process to become visible, we get an execution with very low contention, $k = O(\log\log n)$, but the number of rounds is linear in k. Increasing the number of processes that become visible increases the contention, but decreases the length of the execution. If we allow a constant fraction of processes to become visible, then we get an execution with high contention $k = \Theta(n)$, but the number of rounds decreases to $\log k$.

The extension by one round relies on the fact that only conditional primitives are used. Instead of defining conditional primitives formally, the lower bound proof uses a more refined classification of primitives, according to their *transparency*, defined as follows.

▶ **Definition 3.** Suppose that after an execution prefix α there is a variable v such that $value(v, \alpha) = \perp$ and there is a subset $P = \{p_1, \ldots, p_k\}$ of invisible processes whose next events $\phi_1, \ldots, \phi_k$ apply the same primitive Op to the same variable v:

$$P = \{p_i | p_i \in invisible(\alpha) \wedge next\_event(p_i, \alpha) = \langle p_i, v, Op\rangle\} \ .$$

We say that primitive Op is $(k-m)$-*transparent* (for some $m \leq k$), if there is a permutation π of the next events $\phi_1, \ldots, \phi_k$ such that after execution $\alpha\pi$ at most m processes from P become visible, and the rest of the process in $invisible(\alpha)$ remain invisible.

More formally, define W to be the subset of processes P that become visible after $\alpha\ \pi$:

$$W = \{p_i | p_i \in P \wedge p_i \in visible(\alpha\ \pi)\} \ .$$

The primitive Op is $(k-m)$-transparent if $|W| \leq m$ and every $p_i \in invisible(\alpha) - W$ is in $invisible(\alpha\ \pi)$.

Appendix A shows that read, compare&swap and LL/SC are k-transparent, and that write is $(k-1)$-transparent.

Suppose that the algorithms uses a constant number of primitive types, $Op_1, Op_2, \ldots, Op_t$. Assume, without loss of generality, that each process applies primitives cyclically in this order during its execution. That is, in its i-th step the process performs a primitive of type $Op_{i \bmod t}$. Any algorithm may be modified in this way by introducing "dummy" primitives of the required type. This increases the step complexity of the algorithm by a constant factor, since the algorithm uses a constant number of primitive types, and does not affect the asymptotic step complexity.

For completeness of the explanation, we state Turán's Theorem [23] used in the induction step of the lower bound proof (Lemma 5).

▶ **Theorem 4** (Turán). *Let $G(V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges. If an average degree of G is d, then $G(V, E)$ has an independent set with at least $\lceil |V|/(d+1) \rceil$ vertices.*

The next lemma provides the induction step for the lower bound proof.

▶ **Lemma 5.** *Suppose that there is an execution α_r containing r rounds such that $|invisible(\alpha_r)| = m_r$. Then for any w, $1 \le w \le m_r/2$, there is an execution α_{r+1} containing $r+1$ rounds, such that after α_{r+1} the number of visible processes $|visible(\alpha_{r+1})| = |visible(\alpha_r)| + w$, and the number of invisible processes $m_{r+1} = |invisible(\alpha_{r+1})| \ge 2\sqrt{\frac{m_r \cdot w}{3}} - 2w$.*

Proof. We show how to extend α_r with one round $(r+1)$. In round $r+1$, each process $p_i \in invisible(\alpha_r)$, $1 \le i \le k$, executes its next event $\phi_i = next\_event(p_i, \alpha_r)$. Define round $r+1$ as the sequence of these events $\pi_{r+1} = \phi_1, \phi_2, \ldots, \phi_k$, and define a new execution $\alpha'_{r+1} = \alpha_r\ \pi_{r+1}$.

By assumption, all the next events after α_r apply the same primitive in round $r+1$.

In order to keep many processes invisible, we should take care of two things. First, we need to ensure that the events of round $r+1$ do not conflict with events performed in the previous rounds. Otherwise, if process p in round $r+1$ overwrites a variable previously written by another process q, then p can not be further deleted from the execution without making q visible. We eliminate this kind of conflicts using Turàn's Theorem. Next, we eliminate conflicts between primitives in round $r+1$, using the fact they are k- or $(k-1)$-transparent.

Eliminating conflicts with previous rounds. Consider a *visibility* graph $G(V, E)$, with vertices V corresponding to the processes in *invisible*(α_r). If in round $r+1$, a process p_i accesses one of the variables previously changed by process p_j, then there is an edge $p_i \to p_j \in E$. In round $r+1$, process p_i accesses at most one memory location, and therefore $|E| \le |V|$ and the average degree of G is $d = 2|E|/|V| \le 2$. By Turàn's Theorem, G has an independent set $V' \subseteq V$ with at least $\lceil |V|/(d+1) \rceil = \lceil |V|/3 \rceil$ vertices. We leave the processes corresponding to V' in the execution, and delete all the other invisible processes $V - V'$. That is, we define $\alpha''_{r+1} = \alpha'_{r+1}|_{V'}$. Note that in the execution α''_{r+1}, the processes in V' access only variables which were not changed by other processes. Therefore, there are no conflicts between the primitives of round $r+1$ and the primitives of rounds $1, \ldots, r$.

Eliminating conflicts between events in round $r+1$. Let $M(V')$ the set of variables accessed by the processes in V'. Since $|V'|$ processes access $|M(V')|$ different variables, the average number of processes that access the same variable is $\frac{|V'|}{|M(V')|}$. We order the variables in $M(V')$ by the number of processes accessing them, and choose the w variables $\{v_1, \ldots, v_w\} \in M(V')$ that are accessed by the largest number of processes. In round $r+1$ we keep all the processes accessing the variables $\{v_1, \ldots, v_w\}$, and exactly one process for each variable $M(V') - \{v_1, \ldots, v_w\}$. Let V' be the set of these processes (see Figure 3). We define $\alpha_{r+1} = \alpha''_r|_{V''}$.

For each variable $v_i \in \{v_1, \ldots, v_w\}$, we choose exactly one process $p(v_i) \in V''$ that accesses this variable. In round $r+1$, we schedule the process $p(v_i)$ after all the other processes accessing v_i. Thus, in round $r+1$, process $p(v_i)$ covers all the other processes accessing the variable v_i, $p(v_i)$ becomes visible and is stopped. The rest of the processes $V'' - \{p(v_1), \ldots, p(v_w)\}$ remain invisible after round $r+1$. Below we bound from below the number of processes that can be kept invisible after this round.

Note that the number of processes that access each variable $v_i \in \{v_1, \ldots, v_w\}$ is more than the average $\frac{|V'|}{M(V')}$. Therefore, the number of invisible processes is

$$|V''| - w \ge \frac{|V'|}{|M(V')|} \cdot w + (|M(V')| - w) - w = \frac{|V'|}{|M(V')|} \cdot w + |M(V')| - 2w\ .$$

For simplicity of notation, denote $|M(V')| = x$. Using this notation, the number of processes that are invisible after round $r+1$ is $m_{r+1} \ge \frac{|V'|}{x} \cdot w + x - 2w$.

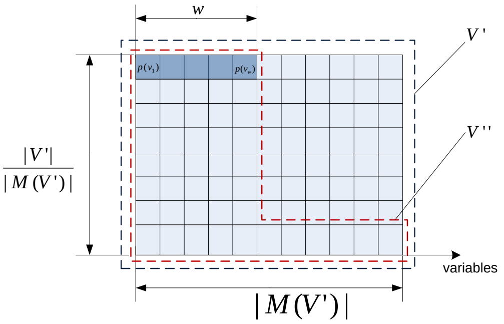


Figure 3 Induction step in the proof of Lemma 5.

Differentiating by x and equating to 0, we get $m'_{r+1}(x) = -\frac{|V'|w}{x^2} + 1 = 0$, implying $x = \sqrt{|V'|w}$. Therefore, the minimal value of m_{r+1} is

$$\frac{|V'|}{\sqrt{|V'|w}} \cdot w + \sqrt{|V'|w} - 2w = 2\sqrt{|V'|w} - 2w \ .$$

Substituting $|V'| = m_r/3$, we get $m_{r+1} \geq 2\sqrt{\frac{m_r \cdot w}{3}} - 2w$. ◀

Different values of w lead to different trade-offs between the total contention and the number of rounds in the execution. For two extreme values of w, we have lower bounds that match the upper bounds:

When $w = 1$, Lemma 5 implies that the number of invisible processes after round $r+1$ is $m_{r+1} \geq 2\sqrt{\frac{m_r}{3}} - 2 \geq \sqrt{\frac{m_r}{3}}$. To prove the lower bound, we need that after r rounds there are at least two invisible processes, i.e., $m_r = 2$. Solving this recurrence, we get that $m_0 = 2^{2^r} 3^{2^r - 1} = \frac{6^{2^r}}{3}$. That is, the total number of the processes in the system should be $n \geq m_0 \in \Omega(6^{2^r})$, or $r = O(\log\log n)$.

In each round of the execution α_r, at most $w = 1$ processes become visible and stopped, and two processes remain invisible after α_r. The rest of processes are deleted from the execution. Therefore the total contention in α_r is $k = r + 2 = \Omega(\log\log n)$. Thus we have an execution with contention $k \in O(\log\log n)$, in which a pair of store and collect operations take at least $\Omega(k)$ steps.

When $w = \frac{|V'|}{2} = m_r/6$, Lemma 5 implies that the number of invisible processes we have after round $r+1$ is $m_{r+1} \geq 2\sqrt{\frac{m_r \cdot w}{3}} - 2w = 2\sqrt{\frac{m_r \cdot m_r/6}{3}} - 2 \cdot m_r/6 = \frac{m_r}{3(\sqrt{2}+1)}$. We need that after r rounds there are at least two invisible processes, i.e., $m_r = 2$. Solving these recurrences, leads to $m_0 = 2\left(3\left(\sqrt{2}-1\right)\right)^r$. Thus, the total number of processes in the system is $n \geq m_0 = 2\left(3\left(\sqrt{2}-1\right)\right)^r$ and hence, $r = \Theta(\log n)$.

In each round r at most $w = m_r/6$ processes become visible and are stopped. Therefore, the total contention in the execution α_r is $k = \sum_i^r \frac{m_i}{6} = \frac{1}{6}\sum_i^r \frac{m_0}{(3(\sqrt{2}+1))^i} = \Theta(m_0) = \Theta(n)$. Thus we have an execution with contention $k \in \Theta(n)$, in which a pair of store and collect operations take at least $r = \Omega(\log k)$ steps.

Thus, we have the following theorem:

▶ **Theorem 6.** *Any implementation of one-time* collect *using $(k-1)$-transparent primitives has an execution of $k+2$ processes in which a pair of* store *and* collect *operations takes at least (a) $\Omega(k)$ steps, provided $k \in O(\log\log n)$, and (b) $\Omega(\log k)$ steps, provided $k \in \Theta(n)$.*

The reduction from collect to $M(k)$-renaming presented in Section 4.1 requires $O(\log M(k))$ additional steps. This implies a linear lower bound on the step complexity of adaptive one-shot renaming with sub-exponential name space $M(k) = 2^{o(k)}$, giving an alternative proof for the lower bound of Alistarh et al. [5].

▶ **Theorem 7.** *An adaptive implementation of one-shot $M(k)$-renaming with sub-exponential name space $M(k) = 2^{o(k)}$ using $(k-1)$-transparent primitives requires at least $\Omega(k)$ steps in an execution with total contention $k \in O(\log \log n)$.*

6 Summary and Open Problems

We have shown that *unconditional* primitives, like fetch&inc, allow snapshot algorithms with $O(\log k)$ step complexity, where k is the total or the point contention. In contrast, when only *conditional* primitives, like compare&swap, are used, any snapshot or collect algorithm must perform $\Omega(k)$ steps, in an execution with total contention $k \in O(\log \log n)$.

We also give an adaptive algorithm whose step complexity is in $O(\min\{k, \log n\})$. The algorithm has a linear step complexity $O(k)$ when contention is low (this is optimal for $(k \in O(\log \log n))$, and logarithmic step complexity $O(\log n)$ when contention is high (this is optimal for $k \approx n$). An immediate question is to understand the complexity of adaptive algorithms in the intermediate range, when the contention is in $o(\log n)$, but still growing faster than $\log \log n$.

It is interesting to investigate whether non-standard *bounded* unconditional primitives, like bounded-fetch&dec, are needed in order to get adaptive algorithm with sublinear step complexity as a function of point contention. We believe that sublinear algorithms adaptive to point contention require unconditional primitives that prevent "wrapping around", like bounded-fetch&dec.

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, September 1993.

2 Yehuda Afek and Yaron De Levie. Efficient adaptive collect algorithms. *Distributed Computing*, 20(3):221–238, 2007.

3 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 262–272. IEEE, 1999.

4 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 71–80. ACM, 2000.

5 Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *J. ACM*, 61(3):18:1–18:51, June 2014. doi:10.1145/2597630.

6 James Aspnes, Hagit Attiya, and Keren Censoe-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of ACM*, 59:2:2–2:24, 2012.

7 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

8 Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.

9 Hagit Attiya and Arie Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. In *Distributed Computing*, pages 149–163. Springer, 2000.

10 Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, February 2002. `doi:10.1137/S0097539700366000`.

11 Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, July 2003. `doi:10.1145/792538.792541`.

12 Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

13 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 217–226. ACM, 2008.

14 Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC'93, pages 91–100, 1993.

15 Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC'04, pages 80–87, 2004.

16 Arie Fouren. *Adaptive Wait-Free Algorithms for Asynchronous Shared-Memory Systems.* PhD thesis, Technion, 2001.

17 Maurice Herlihy, Victor Luchangco, and Mark Moir. Space-and time-adaptive nonblocking algorithms. *Electronic Notes in Theoretical Computer Science*, 78:260–280, 2003.

18 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

19 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 285–294. ACM, 2003.

20 Prasad Jayanty. f-arrays: Implementaion and applications. In *Proceedings of the 21th Annual Symposium on Princeples of Distributed Computing (PODC)*, pages 270–279, New York, 2002. ACM.

21 Yong-Jik Kim and James H Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

22 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995.

23 P. Turan. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

A Transparent Primitives

▶ **Claim 8.**

(a) *the* write *primitive is* $(k-1)$*-transparent;*

(b) *the* read *primitive is* k*-transparent;*

(c) *the* compare&swap *primitive is* k*-transparent;*

(d) *the* LL/SC *prmitives are* k*-transparent.*

Proof. According to definition 3, suppose that after an execution prefix α there is a variable v such that $value(\alpha) = \bot$ and there is a subset $P = \{p_1, \ldots, p_k\} \subseteq invisible(\alpha)$ whose next events $\phi_1, \ldots, \phi_k$ contain the same primitive Op on the variable v:

$$P = \{p_i | p_i \in invisible(\alpha) \wedge next\_event(p_i, \alpha) = \langle p_i, v, Op \rangle\}$$

(a) *Op* **is a** write. Let $\pi = \langle \phi_1, \ldots, \phi_k \rangle$. In events $\phi_1, \ldots, \phi_k$ processes P do not read any information, therefore processes *invisible*$(\alpha) - P$ remain invisible also after $\alpha \circ \pi$. All the write events $\phi_1, \ldots, \phi_{k-1}$ are overwritten by the following writes, thus these events are undetectable, and therefore processes $p_1, \ldots, p_{k-1}$ remain invisible also after $\alpha \circ \pi$. (The only process that becomes visible after $\alpha \circ \pi$ is p_k that performs the last write event in π. Since this event overwrites the previous values written to v, it can not be deleted from the execution undetectably). By definition 3 this implies that write is $(k-1)$-transparent.

(a) *Op* **is a** read. Define a permutation $\pi = \langle \phi_1, \ldots, \phi_k \rangle$. Since events ϕ_i read from an empty variable v and do not change values of other variables, there is no information flow between the processes in *invisible*(α), and therefore all the processes invisible after α remain invisible also after $\alpha \circ \pi$. According to definition 3 this imply that read is k-transparent.

(c) *Op* **is a** compare&swap We define the permutation π of the events $\phi_1, \ldots, \phi_k$ as follows. As mentioned in the introduction, we assume that no process attempts to write $\perp$. First, we schedule primitives CAS(u, w) where $u \neq \perp$. By semantics of CAS these primitives fail and do not change the value of v. Then we schedule the remaining primitives CAS$(\perp, w)$, where $w \neq \perp$. By atomicity of CAS, only the first of these primitives reads $v = \perp$ and succeeds, while the rest read $v = w \neq \perp$ and fail.

Since none of the events π performed by processes P reads any value written previously by another process, therefore all the processes in *invisible*$(\alpha) - P$ remain invisible after $\alpha \circ \pi$. The $k-1$ unsuccessful CAS primitives do not change the value of v. Since a successful CAS (that changes the value of v from $\perp$ to w) does not overwrites any value previously written to v by another process, all the events π by processes P can be removed from the execution without affecting the rest of the invisible processes *invisible*$(\alpha) - P$. Therefore all, the processes P remain invisible after $\alpha \circ \pi$, implying *invisible*$(\alpha \circ \pi) =$ *invisible*(α). By definition 3, CAS is k-transparent.

(d) *Op* **is an** LL. Define $\pi = \phi_1, \ldots, \phi_k$. All these primitives read $\perp$ from v, thus and they do not get any information written previously by other invisible processes. Therefore all the processes *invisible*$(\alpha) - P$ remain invisible also after $\alpha \circ \pi$. Since LL primitives do not overwrite any value previously written to v by other processes, all the events π by processes P can be removed from the execution without affecting the steps of other invisible processes. Therefore, *invisible*$(\alpha \circ \pi) =$ *invisible*(α). By Definition 3, LL is k-transparent.

Op is an SCl. Let $\pi = \phi_1, \ldots, \phi_k$. By the semantics of LL/SC, only the first SC, corresponding to event ϕ_1 succeeds, and the rest fail. Since SC does not read any value previously written by another process to v, the processes *invisible*$(\alpha) - P$ remain invisible also after $\alpha \circ \pi$. The $k-1$ unsuccessful primitives do not change the value of v, and the single successful

$SC(w)$, $w \neq \perp$ does not overwrite any value written previously by another process. Therefore, all the computational events π can be removed from the execution without affecting the values of other invisible processes. Thus, all the processes P remain invisible after $\alpha \circ \pi$. By Definition 3, SC is k-transparent. ◀